Dipartimento di Informatica
Università del Piemonte Orientale "A. Avogadro"
Spalto Marengo 33, 15100 Alessandria
http://www.di.unipmn.it

Making CORBA fault-tolerant

Author: Daniele Codetta Raiteri
(daniele.codetta_raiteri@unipmn.it)

TECHNICAL REPORT TR-INF-2004-04-07-UNIPMN
(April 2004)

The University of Piemonte Orientale Department of Computer Science Research Technical Reports are available via WWW at URL
`http://www.di.mfn.unipmn.it/`.
Plain-text abstracts organized by year are available in the directory

Recent Titles from the TR-INF-UNIPMN Technical Report Series

2004-06 Orthogonal operators for user-defined symbolic periodicities, Egidi, L., Terenziani, P., April 2004.

2004-05 RHENE: A Case Retrieval System for Hemodialysis Cases with Dynamically Monitored Parameters, Montani, S., Portinale, L., Bellazzi, R., Leonardi, G., March 2004.

2004-04 Dynamic Bayesian Networks for Modeling Advanced Fault Tree Features in Dependability Analysis, Montani, S., Portinale, L., Bobbio, A., March 2004.

2004-03 Two space saving tricks for linear time LCP computation, Manzini, G., February 2004.

2004-01 Grid Scheduling and Economic Models, Canonico, M., January 2004.

2003-08 Multi-modal Diagnosis Combining Case-Based and Model Based Reasoning: a Formal and Experimental Analisys, Portinale, L., Torasso, P., Magro, D., December 2003.

2003-07 Fault Tolerance in Grid Environment, Canonico, M., December 2003.

2003-06 Development of a Dynamic Fault Tree Solver based on Coloured Petri Nets and graphically interfaced with DrawNET, Codetta Raiteri, D., October 2003.

2003-05 Interactive Video Streaming Applications over IP Networks: An Adaptive Approach, Furini, M., Roccetti, M., July 2003.

2003-04 Audio-Text Synchronization inside mp3 file: A new approach and its implementation, Furini, M., Alboresi, L., July 2003.

2003-03 A simple and fast DNA compressor, Manzini, G., Rastero, M., April 2003.

2003-02 Engineering a Lightweight Suffix Array Construction Algorithm, Manzini, G., Ferragina, P., February 2003.

2003-01 Ad Hoc Networks: A Protocol for Supporting QoS Applications, Donatiello, L., Furini, M., January 2003.

2002-06 Stochastic modeling, analysis techniques and tools for dependable reactive systems, Codetta Raiteri, D., Bobbio, A., October 2002.

2002-05 Stochastic modeling, analysis techniques and tool for dependable reactive systems, Bernardi, S., Gribaudo, M., Bobbio, A., October 2002.

2002-04 Interactive MPEG video streaming over IP-Networks: a performance report, Furini, M., Roccetti, M., September 2002.

2002-03 A fuzzy approach to case-based reasoning through fuzzy extension of SQL, Portinale, L., Montani, S., Bellazzi, R., July 2002.

Contents

Introduction

A large computer network (as Internet or a corporate Intranet) is typically *heterogeneous* meaning that it connects workstations, servers or personal computers with different hardware architectures or operating systems; at the same time, such a net may be composed by several subnets with various network protocols and the applications having to interact to compose a distributed system may be written in different (object-oriented) programming languages.

Developing distributed systems on *heterogeneous* networks is a hard task; the *Object Management Group* (**OMG**) was founded in 1989 and its purpose is the creation of standards to achieve interoperability and portability of distributed object-oriented applications in a heterogeneous environment; OMG produces specifications gathering the ideas of the OMG members responding to the *Requests For Information* (**RFI**) and the *Request For Proposals* (**RFP**) issued by OMG.

Facilities for distributed object-oriented computing are defined by the OMG *Object Management Architecture* (**OMA**); the core of the OMA is the *Object Request Broker* (**ORB**) that provides transparency of objects location, activation and communication: the ORB hides low-level details about platforms and networks interfaces to the developers, so they can focus on the their applications without dealing with distributed computing infrastructures.

The *Common Object Request Broker Arhitecture* (**CORBA**), published for the first time in 1991 is a concrete description and specification of services and interfaces that ORBs must provide; despite its original flexibility to integrate several heterogeneous applications in a distributed system, CORBA has had to evolve during the years introducing new features; *fault tolerance* is one of the evolutive directions and *Fault Tolerant CORBA* (**FT-CORBA**) is the standard resulting from the contributes of OMG members in that sense.

A distributed application is said to be *fault tolerant* if it can be properly executed despite the occurence of faults; distributed applications such as e-commerce or air traffic control need high reliability and therefore they require fault tolerance; in CORBA version 2.6, fault tolerance is explicitly addressed

for the first time.

This paper first describes the general architecture of CORBA standard (**Chapter 1**); then it focuses on the ways to make CORBA fault tolerant exploring FT-CORBA specifications (**Chapter 2**) and providing the case of a real system (**DOORS**) (**Chapter 3**).

Chapter 1

The Object Management Architecture

1.1 Introduction to OMA

In the OMA object model [1], objects provide services performed when clients issue requests; the OMA is composed of:

- **Object Model** - it defines the way to describe the objects distributed across a heterogeneous environment: the identity of an object is immutable and the service it provides can be accessed only through a specific *interface* whose definition must respect particular rules; the location and the implementation of an object is transparent to the client requesting the corresponding service.
- **Reference Model** - it defines how objects must interact: Fig. 1.1 [2] shows the scheme of the OMA Reference Model where ORB is the key component of such a structure; ORB allows the communications between clients and objects (servers) and four object interface categories are available:
  - **Object Services** - they are domain indipendent interfaces necessary for services such as objects lifecycle management, security, transactions, event notifications and discovery of other available services; in the last case, the Object Services are:
    - *The Naming Service* - it allows clients to search objects (servers) by name;
    - *The Trading Service* - it allows clients to search objects (servers) by their properties.

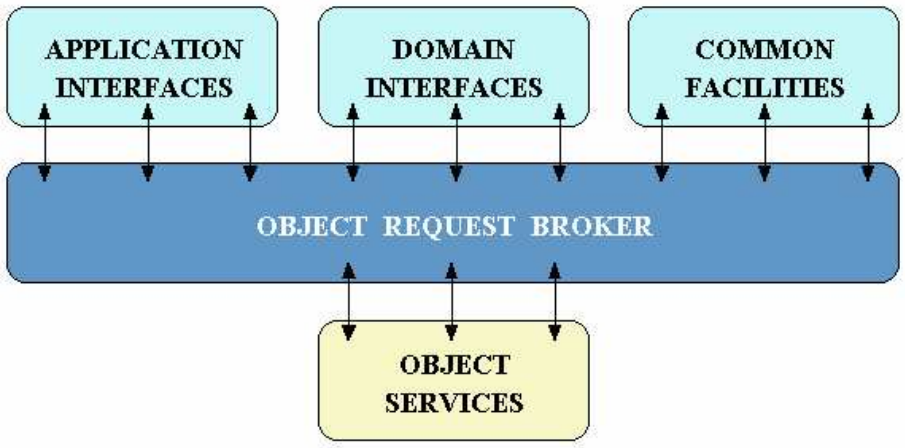

Figure 1.1: OMA Reference Model Interface Categories

- **Common Facilities** - they are interfaces for end users applications;
- **Domain Interfaces** - they are oriented towards specific application domains; several separate application domains may be present at the same time;
- **Application Interfaces** - they are application-specific interfaces and for this reason OMG does not define any standard for them.

Fig. 1.2 [3] shows the **Object Frameworks** of the Reference Model; they are groups of interacting objects; each group is domain-specific and provides services for that domain; peer-to-peer communication is estabilshed among the objects within a group: a component of a group may be a server or a client.

Any other role, different from those mentioned above, are provided by the ORB, the core of the OMA.

1.2 The Common Object Request Broker

The main component of the OMA is the ORB whose details are defined by the CORBA [4] [5] [3] [6] specification; the ORB is composed by:

- ORB core
- OMG Interface Definition Language (OMG IDL)

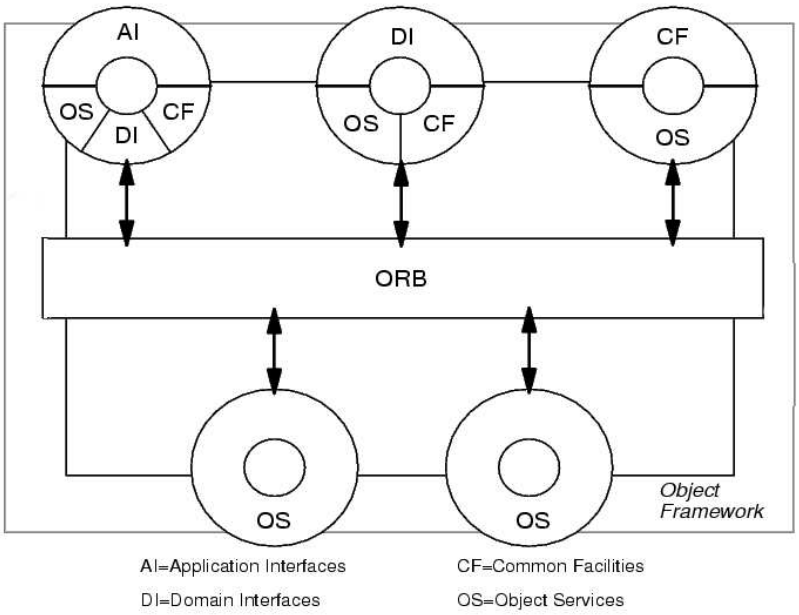

Figure 1.2: OMA Reference Model Object Framework

- Language Mappings
- Stubs and Skeletons
- Interface Repository (IR)
- Dynamic Invoaction and Dispatch
- Object Adapters (OA)

The following subsections describe each of the CORBA features. Fig. 1.3 [2] shows the CORBA architecture.

1.2.1 The ORB core

In the OMA model, objects provide services; the purpose of the ORB core is the delivery of the requests (issued by the clients) to the objects (servers) and returning the corresponding output results to the clients; the way the ORB accomplish this purpose must be completely transparent to the clients: they do not have to know about the following objects features:

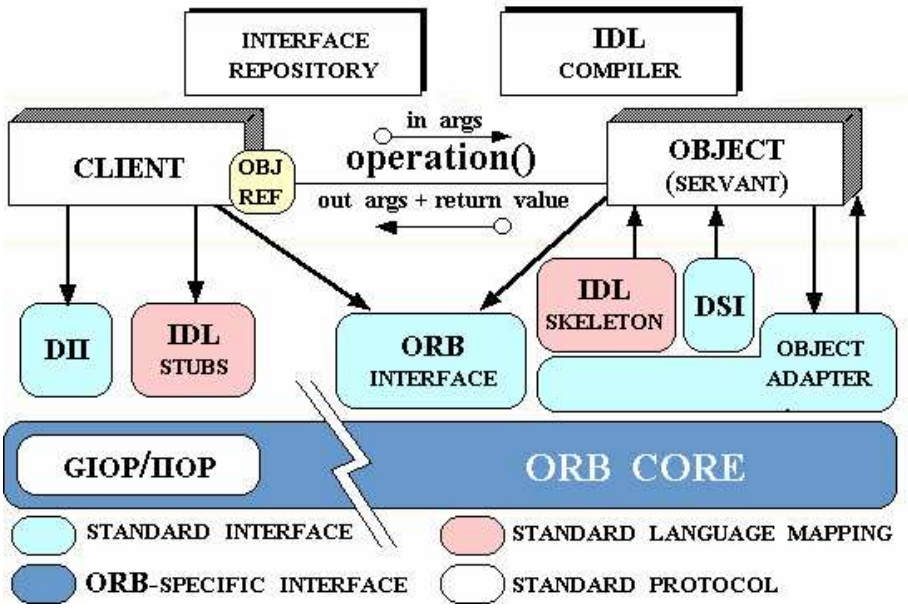

Figure 1.3: CORBA architecture

- **Object location** - the client does not know about the object location; it may be anywhere: on the same machine of the client process or on another machine in the network;
- **Object implementation** - the client does not know about the programming language the object has been implemented, on which operating system and hardware the object is running;
- **Object execution state** - when the client make a request to the *target object* (the object providing the service needed by the client), it does not know about the object *state*: the object may be activated or not; if the target object is not ready to receive requests, the ORB must activate it in a transparent way before delivering to it the request issued by the client;
- **Object communication mechanisms** - the client does not know about the communication mechanisms the ORB uses to deliver the request to the target object and to return the output results to the client.

When a CORBA object is created, an **object reference** is specifically created for it; an object reference is *immutable* meaning that it always refer

to the same object and *opaque* meaning that a client can access to an object reference to identify the target object, but it can not modify the object reference. An object reference is about a single object and exists as long as the object exists: in this sense, we can distinguish between:

- *persistent* objects - they live beyond the process in which they are created or activated;
- *transient* objects - they live as long as they are necessary; they are useful in situations requiring temporary services.

When a client has to make a request, the client specifies the target object by means of the object reference: how can the client get the object reference for the service it requires? That is possible by means of the *Directory Service* (**DS**): it consists of storing existing objects information and references; when a client has to obtain the object reference for the service it needs, the client invokes the DS to perform a *lookup* by name or by properties of the target object and the DS returns to the client the reference to the target object, if found.

To avoid several invocations of the DS (supplied by an object) for the same service by the same client, an application can ask the ORB to convert an object reference to a string; the client can store the string into a file or database; when the client needs the object referencence for that service, the client asks the ORB to turn the string back to a reference. This capability allows to keep persistent links between objects and clients.

In this way also the ORB can supply object references allowing clients to get them directly from the ORB without accessing to the DS. The ORB works as a DS even when a client needs the reference of the object performing the DS: when a client needs an object reference for a service for the first time in its life, it does not know neither the reference of the required service nor the reference of DS. In this case, the client first gets the object reference of the DS from the ORB, then the client makes a request to the DS in order to obtain the object reference of the required service.

Anyway the key role of the ORB is the communication between clients and objects: the CORBA standard tries to keep the ORB as simple as possible assigning most of the functionalities to the other OMA components.

1.2.2 OMG Interface Definition Language (OMG IDL)

A client must know the types of operations an object supports; for this reason, an **object interface** is defined for each object to inform about the the operations the object can perform, the arguments it needs and their

types, the returned values and their types. Object interfaces are defined using the OMG *Interface Definition Language* (**OMG IDL**) and are similar to C++ classes and Java interfaces.

Object interfaces are defined separately from object implementations: in this way, object can be implemented using several programming languages while they can communicate with each other using a unique declarative language, the OMG IDL. We can say that the OMG IDL is *language indipendent*.

OMG IDL provides several data types we can find in other programming languages such as *long*, *double* and *boolean*; it provides structured types too, such as *struct* and *union*; another feature of OMG IDL are the *exceptions*, useful to deal with exceptional conditions during the execution of applications.

Another important feature of the OMG IDL is the **Interface Inheritance**: an interface can inherite parameters and methods from one or more interfaces; in this way, existing interfaces can be reused or extended when defining new services.

1.2.3 Language Mapping

The OMG IDL is not a programming language, but a declarative language; the way to map the OMG IDL features to a given programming language is provided by the **language mappings**; the OMG has standardized language mappings for several programming language such as C++, Java and Smalltalk.

The language mappings are the "point of contact" between CORBA object interfaces and the "real world" of implementation, so a language mapping must be complete to allow programmers to be able to respect the CORBA standard using that programming language. For this reason, the language mappings are periodically improved to face the evolution of the programming language and the requirements of new features.

1.2.4 Stubs and Skeletons

The conversion of object interfaces from IDL to a specific programming language generates client-side **stubs** and server-side **skeletons**: a stub is a mechanism to create and issue requests, while a skeleton is a mechanism to deliver requests to the objects; stubs and skeletons are interface specific since they are directly built in the client and server implementation, so they need a complete knowledge of the object interface they are associated to.

The request of services by means of stubs and skeletons is called **static invocation**: the stub converts the client request from its representation in

the programming language to one suitable to be trasmitted; the request in this form is passed to the ORB which delivers it to the target object; on the server-side, the skeleton converts the request from the transmitting form to the target object programming language representation.

At this point the target object can work on the request and when the response is ready, it is converted in a transmitting form by the skeleton, it is passed to the ORB which moves the response to the client whose stub performs another conversion of the response, from the transmitting form to the client programming language representation. Fig. X shows this scheme.

1.2.5 Interface Repository (IR)

A client must know the interface of the object providing the desired service; a direct way to get that information consists of compiling or translating the object interface into the code of the client application respecting the mapping rules for that programming language defined by a specific language mapping.

This approach is not convenient when some changes to the distributed system are applied in a way that object interfaces have to be modified; in this case, the client application must be partially rebuilt and recompiled. To avoid this problem, the CORBA *Interface Repository* (**IR**) has been specified in the CORBA architecture with the purpose of allowing the access and the update of IDL system at runtime.

The IR is an object that can be invoked by clients to obtain the interface of other objects; the IR provides the interface for the requested object by traversing the hierarchy of IDL information that IR stores. This role is fundamental for another feature of CORBA called *Dynamic Information Interface* (DII) (*section 1.2.6*).

1.2.6 Dynamic Invocation and Dispatch

Stubs and skeletons allow to invoke operations on known objects (static invocation), but some applications may need the invocation of operations supplied by some objects without having compile-time knowledge of their interfaces (**dynamic invocation**).

CORBA supports dynamic invocation too, introducing the *Dynamic Invocation Interface* (**DII**) for the dynamic client request invocation and the *Dynamic Skeleton Interface* (**DSI**) for the dynamic dispatch to the object. The DII and the DSI replace stubs and skeletons in the dynamic invocation; we can consider the DII and the DSI as a generic client-side stub and a generic server-side skeleton respectively: they are able to invoke and dispatch any request to any object without knowing a priori its interface.

When a client has to make a request for a service using DII, these are the steps to follow: the client has to obtain the target object interface by using the IR; from the interface, the client obtains the information about operations supported by the object and the relative arguments; then, the client has to create a **Request Object** containing the arguments for the desired operation respecting the target object interface; at this point the Request Object is passed to the ORB for the delivery to the target object.

Comparing static invocation with DII, we can say that the invocation of a DII request could require several remote invocations (to the ORB and the IR), making a DII request more expensive than its equivalent static invocation which does not suffer of the overhead of accessing to the IR.

While the DII allows clients to make invocations without having a stub, on the server-side, the Dynamic Skeleton Interface (DSI) allows object providing services to receive requests without an interface-specific skeleton; the DSI is provided by the server application POA (*section 1.2.7*) and translates the Request Object to the programming language format before passing it to the object implementation; as the DII, DSI can access to the IR to get the information about target object IDL interface.

1.2.7 Object Adapters (OA)

Object Adapters (**OA**) mediate between the CORBA object environment and the implementations (*servants*) in a specific programming language; a servant is said to "incarnate" a CORBA object, meaning that a servant is the implementation of the CORBA object in a given programming language; for instance, in C++ or Java, servants are instances of classes.

An OA is a "link" between CORBA object interface to the real object interface specified in the programming language implementation; CORBA version 2.2 introduced the *Portable Object Adapter* (**POA**) providing the complete specifications for OAs and replacing the OMG *Basic Object Adapter* (OMG BOA) which suffered of several ambiguities, missing features and problems of portability. Fig. 1.4 [3] shows the role of the POA consisting of the following facilities:

- **Object Creation** and **Reference Generation** - a programming language entity can be registered in the CORBA environment as a CORBA object by means of the POA: the server application asks the POA to create a new object and the POA returns an object reference which uniquely indentifies that object in the CORBA environment; when the object has been created, its interface is passed to the IR to be stored.

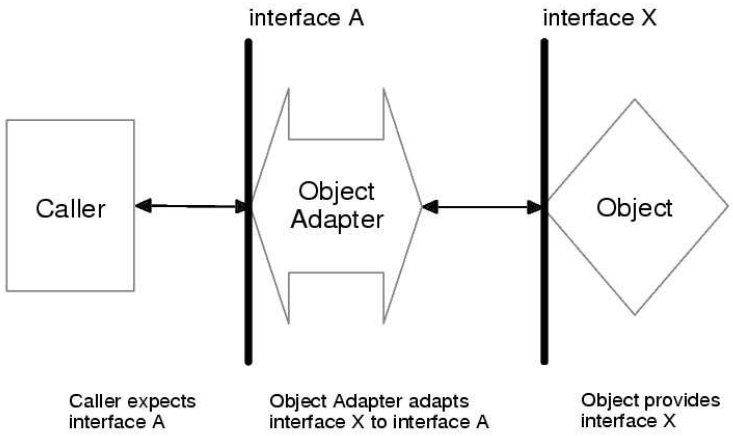

Figure 1.4: Role of an OA

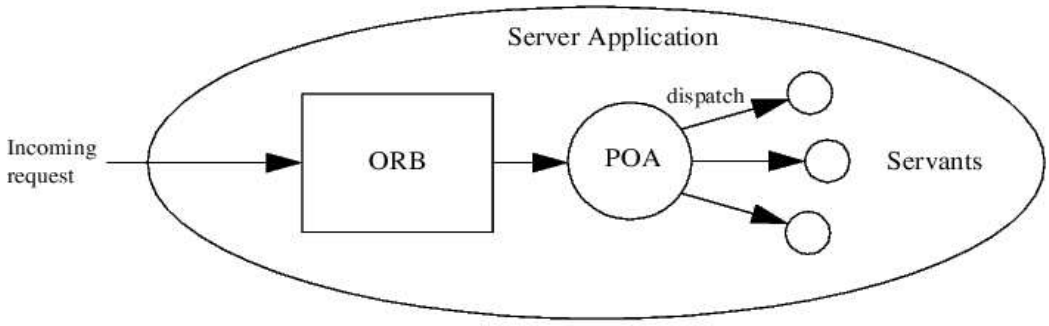

Figure 1.5: Request flow from client to servant through POA

- **Object Upcalls** - POA allows the dispatch of the requests for a service to the servant implementing that service; let's consider a request flowing from the client to the server as in Fig. 1.5 [6]: the client invokes a request using the target object reference; the target object will be contained in a server application. Since a server application may be composed by several modules implemented in serveral programming languages, the server application may have several POAs associated to it: the ORB receives the request from the client and by means of the **object key** contained inside the target object reference, dispatches the request to the correct POA; at this point the selected POA converts the invocation arguments to the format of the relative programming language and by means of the **object ID** contained inside the object key, dispatches the request to the correct servant implementing that service; then, when the output result is ready, it is delivered back to the POA, to the ORB and finally to the client. To summarize, POA connects the world of CORBA object with the "real world" of programming language implementations.

- **Object Activation** - POA can activate single objects when a request for them has arrived and they are not active; this is necessary when an application with many CORBA objects may only want to activate those objects that actually receive requests, minimizing the resources usage; the associations between object IDs and servants are stored inside the POA by the **Active Object Map**: a single servant may "incarnate" one or more CORBA objects; this is useful when an application hosts a large number of objects and it would be difficult to deal with a large number of servants; if instead, the application hosts a small number of objects with the same IDL interface type, there may be a unique servant (*default servant*) implementing every object hosted by that application.

1.3 CORBA messaging

Clients can invoke operations on target objects in three different *synchronous* ways:

- **Synchronous Invocation** - the client invokes the request, then it gets blocked waiting for the response; this approach is equivalent to a *Remote Procedure Call* (RPC);

- **Deferred Synchronous Invocation** - the client invokes the request, then it goes on with its own processing, collecting the response later,

when it arrives; this approach is useful when the client has to invoke several services requiring a long time running; in this way the client can make the invocations parallelly and collect the responses as they are ready;

- **Oneway Invocation** - the client invokes the request, continues its processing and does not wait for the response; the ORB only guarantees that the request will be delivered to the target object, but it does not guarantee the delivery of the response back to the client; the target object might invoke a *callback* on the client to advertise that the request has been completed successfully; this approach was introduced in order to allow a safe delivery of requests using unreliable transport protocols such as *User Datagram Protocol* (UDP).

Synchronous messaging assume that the communication is performed on a reliable connection, but in the "real world" disconnections may occur in a large-scale distributed system; for this reason, OMG issued the CORBA **Messaging Specification** [7] introducing asynchronous messaging techniques, time-indipendent invocation and specifications for messaging Quality of Service (QoS); 3.0 is the first version of CORBA to respect the CORBA Messaging Specification which defines two asynchronous request techniques:

- **Callback** - the client supplies in the Request Object its own reference and when the response is ready , the ORB uses the client reference to deliver the responce to the client;

- **Polling** - when the client invokes an operation, a *valuetype* (*section 1.4*) is immediately returned; the valuetype can be used by the client to poll or wait for the response indipendently.

1.4 Objects by value

In CORBA, the arguments passed as arguments to operations may be data types such as integers, structs or array; CORBA version 3.0 introduced the possibility to pass references to objects as arguments: a new contruct called **valuetype** [8] has been added to OMG IDL; it is a cross between a struct and a interface (similar to C++ and Java class definitions) supporting data members, methods and single inheritance from another valuetype.

When a valuetype is passed as an argument to a remote operation (performed by another object), the receiver creates a copy of the object in its addressing space; any operation performed on such a copy does not modify

the original object on the client-side: all operations invoked on valuetypes are always performed locally to the process.

1.5 Inter-ORB protocols

Before CORBA version 2.0, different ORB systems could not interoperate because CORBA did not specify any rule for the communication between two ORBs; CORBA version 2.0 introduced **direct interoperability** and **bridge-based interoperability**: we have direct interoperability when ORBs reside on in the same *domain* meaning that they understand the same object references and the same OMG IDL type system; if ORBs reside on different domains, bridge-based interoperability is necessary; the role of bridges is the mapping of information from an ORB domain to another.

ORB interoperability is generally defined as a protocol by the *General Inter-ORB Protocol* (**GIOP**), while the *Internet Inter-ORB Protocol* (**IIOP**) provides the specifications about the GIOP implementation using TCP/IP: IIOP establishes that hosts must be connected by a common network based on a point-to-point protocol.

In a ORB interoperability context, CORBA defines an object reference format called *Interoperable Object Reference* (**IOR**) which stores the information needed to locate an object in such a context and communicate with it.

Chapter 2

Fault Tolerant CORBA

2.1 Introduction to FT-CORBA

A distributed application is said to be *fault-tolerant* if it can be properly executed despite the occurence of faults; the main way to obtain fault-tolerance is by *server replication*: if a service is supplied by several server *replicas*, instead of one, the fault of one of them does not compromize the distributed system because that service can be supplied by another replica.

In its first versions, CORBA did not provide any strategy to achieve reliability of applications in a distributed object-oriented environment; for this reason, developers added replication in their CORBA systems (DOORS [9], Eternal [10], AQuA [11], etc.) to cope with object failures and in 1998, OMG issued a RFP producing in 2000 the *Fault Tolerant CORBA* (**FT-CORBA**) [12] [13] [14] specification which embeds many ideas implemented in the systems mentioned above. In CORBA version 2.6, fault tolerance is explicitly addressed for the first time.

Mechanisms allowing CORBA to be fault-tolerant are built on top of standard CORBA with minimal modification to existing ORBs preserving OMA *Object Model* and *Object Reference Model* (*section 1.1*).

2.2 Replication Logic

Replication Logic is a set of protocols to rule the way distributed systems have to handle server replication; two replication techniques are defined:

- **active replication** - the client requires a service to a set of deterministic replicas waiting for a certain number of identical replies; this number depends on the type of fault a replica may suffer of, and by

the *consistency* [15] criterion which must be verified on the returned results; for instance, the majority of replies must be identical;

- **passive replication** (or *primary-backup approach*) - the client requires a service to a particular replica, the *primary replica*, while every other replica is considered as a *backup replica*; if the primary replica fails, the backups elect a new primary replica to process the requests.

Both techniques require mechanisms to work properly such as a reliable multicast protocol, a failure detection system and an agreement protocol; these mechanisms allow to implement the notion of *group abstraction* (a set of members working as a single entity) with the relative services such as membership, state transfer, etc. The multicast protocol is used for the communication among group members, the failure detector discovers the failures inside the group and the agreement protocol allows the election of a new primary, if necessary.

2.2.1 Intrusiveness

Depending on the degree of *intrusiveness* (Fig. 2.1 [14]) of the replication logic with respect to the standard ORB, the FT-CORBA system design can be classified as:

- **Intrusive design** - it requires to embed a part (or all) of the replication logic inside the ORB; the intrusiveness may be *deep* or *shallow* depending on the number of replication logic component embedded inside the ORB;

- **Non-intrusive design** - the replication logic is separate from the ORB and may be *"above"* (Fig. 2.2 [14]) or *"below"* (Fig. 2.3 [14]) the ORB: in the first case, the replication logic exploits only ORB features, in the second case, at least one mechanisms used by the replication logic is not provided by the ORB and requests pass through an **Interception Layer** whose role is adapting the requests to the format required by the adopted replication logic.

2.2.2 Interoperability

Interoperability (Fig. 2.4 [14]) means the possibility of run-time interactions of objects deployed on top of distinct ORBs; the replication logic may be able to guarantee interoperability or not; a CORBA system is interoperable only if it uses the remote method invocations offered by the ORB via IIOP

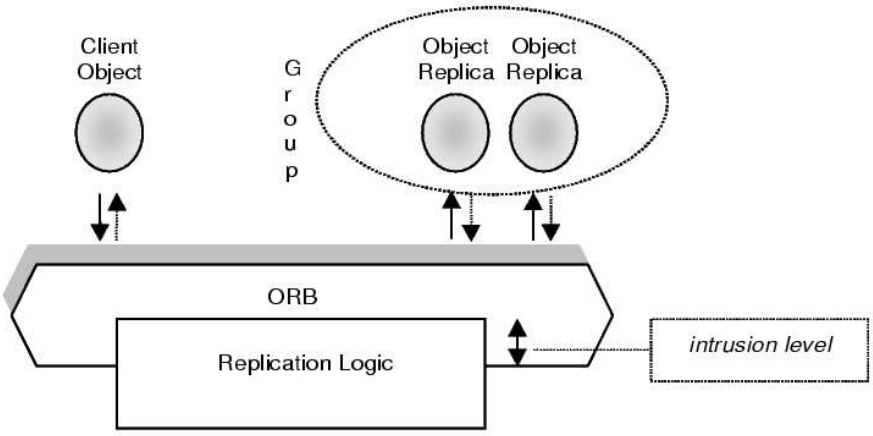

Figure 2.1: Intrusion level of the replication logic

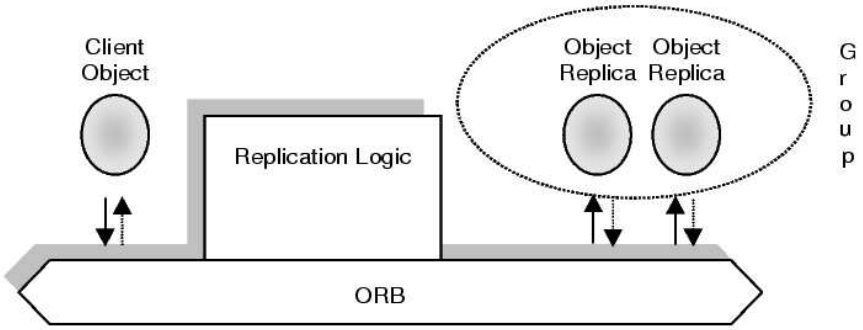

Figure 2.2: Replication logic above the ORB

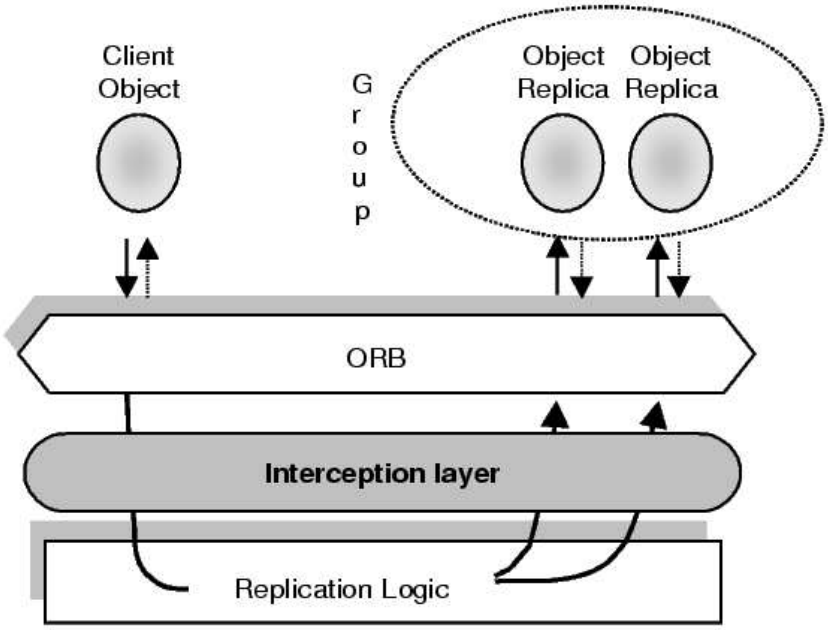

Figure 2.3: Replication logic below the ORB

(*section 1.5*); to allow interactions between already existing ORBs, they may need some modifications or not; for instance, if the desing is non-intrusive and developed above the ORB, the system is interoperable without any modification on the ORB.

In other system configurations, the system may not be interoperable; for instance, in a system it may be necessary to provide the same ORB to all computing resources.

2.3 FT-CORBA architecture

2.3.1 Object groups

In the FT-CORBA architecture (Fig. 2.5), fault tolerance is achieved by **object replication**, **fault detection** and **recovery**; all the components in the CORBA architecture are implemented as CORBA objects using IDL interfaces, servants, etc. (**Chapter 1**).

Using object replication, *consistency* [15] among replicas is achieved managing replicated objects as an **object group**: a set of replicas of the object providing the service, accessed by clients as a single entity; an object group

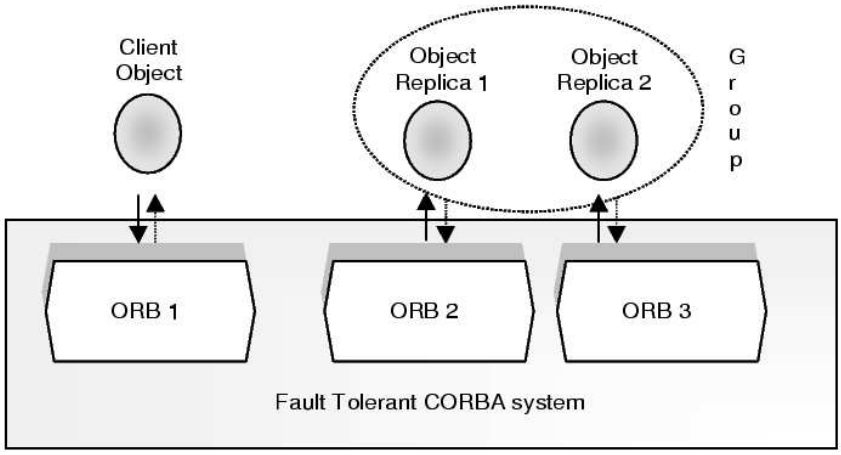

Figure 2.4: Interoperability in a multi-ORB environment

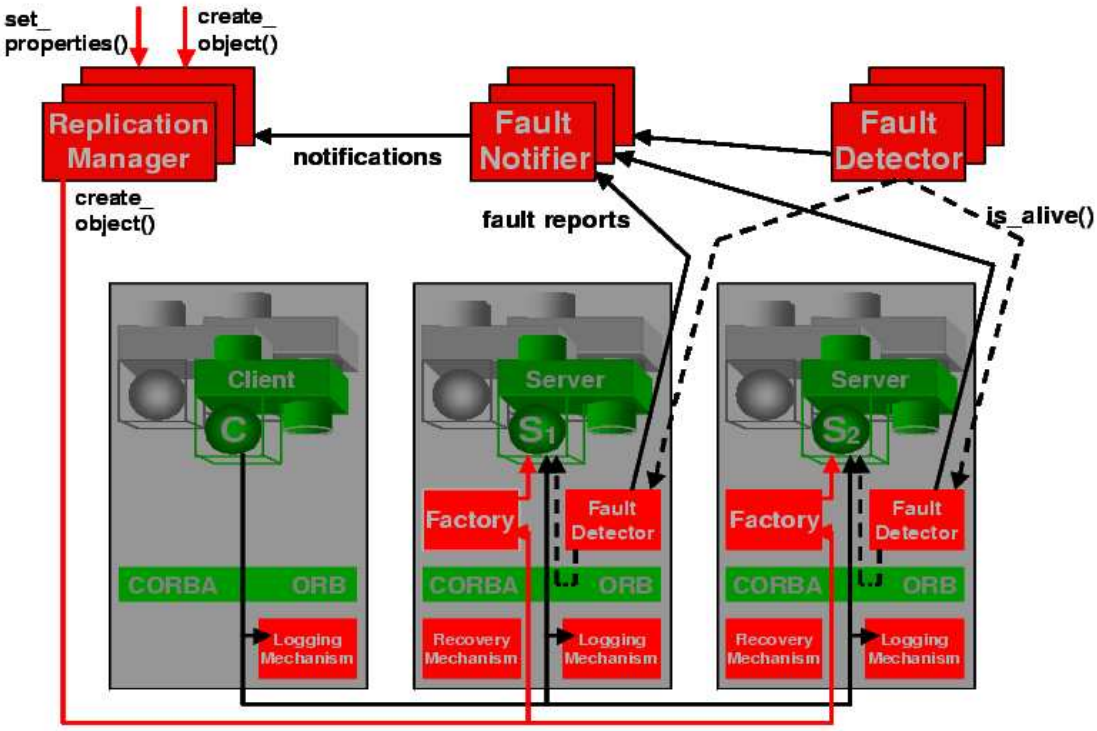

Figure 2.5: FT-CORBA architecture

is *transparent* to the client with two levels:

- **replication transparency** - clients are not aware of object replication: the object group offers the same interface as the replicas it contains and the client code used to bind an object group is the same as for binding a single object; the client sends a single request and receives a single reply;
- **failure transparency** - clients are not aware of object failures and of the reliability protocols.

Object replication is applied to the application objects and also to the components of the FT-CORBA infrastructure providing fault tolerance; the mechanisms defined by FT-CORBA to handle object groups are the following:

- **Interoperable Object Group Reference (IOGR)** - while an IOR (*section 1.5*) identifies an object and contains the information (**profile**) allowing the ORB to establish a network connection with the object, an IOGR identifies an object group and contains the IORs for all the object group members. ORB uses IOGR to establish a network connection with the object group, transparently to the client. Fig. 2.6 shows the structure of a generic IOGR: for every object group member, a IIOP profile is present; each profile contains the host, the port and the object key to reach the target object, and one **TAG_GROUP** component which is composed by four fields allowing to uniquely identify an object group:
  - *tag_group_version* - it indicates the version of this tagged component;
  - *ft_domain_id* - it is the identifier of the Fault Tolerance Domain (explained in this section) the IOGR belongs to;
  - *object_group_id* - it is the identifier of the object group and it is assigned by the Replication Manager (*section 2.3.2*) when an object group is created;
  - *object_group_version* - it indicates the version of the object group membership which may change during the time; this field is useful when the IOGR used by the client must be updated.

  The **TAG_FT_PRIMARY** component informs the ORB about the primary member of the object group in the case of passive replication; the **Multiple Component Profiles** contains information about the object group such as the number of members.

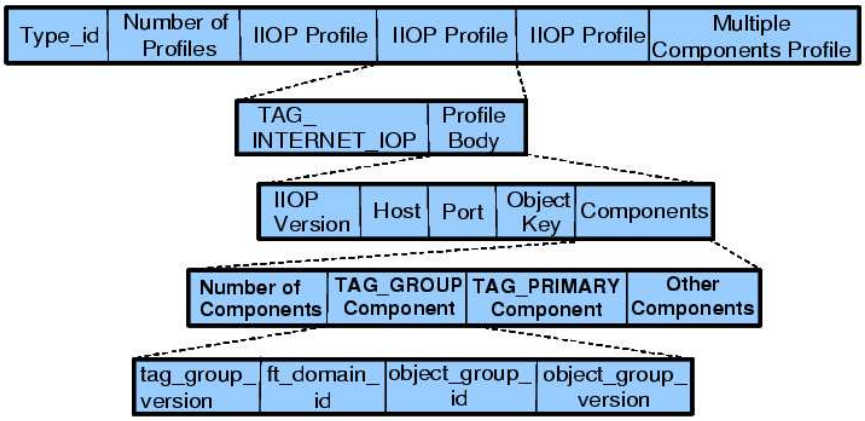

Figure 2.6: IOGR structure

- **Transparent client reinvocation and redirection** - when a client needs a service, it connects the ORB passing the IOGR of the corresponding object group; then, the ORB passes the request to one of the members of the group; if the invocation of the target object fails, the request will be redirected to another object of the same group by the ORB; in any case, the client requires the service only once. The client may receive an exception from the ORB if every object group member has failed. The membership of a group may change during the time, so in the IOGR passed by the client to the ORB, the version of the object group is included; if the ORB detects that the object group version is obsolete, it provides the client with the updated IOGR and the client will request the service to the correct object group reference. An ORB must consider an invocation to a replica as concluded when:

  - the ORB has received the reply;
  - the request expiration time has elapsed;
  - the ORB has received an exception from the object.

  Only in the first case, the ORB does not redirect the request to another object group member. If the client and the target object refer to distinct ORBs, two ORBs participate to the mechanism (*section 1.5*); in this case, the client passes the request to its ORB (**client ORB**) which delivers the request to the target object reading IOGR information; when the reply is ready, the target object passes the reply to its ORB (**server ORB**) which delivers the reply back to the client.

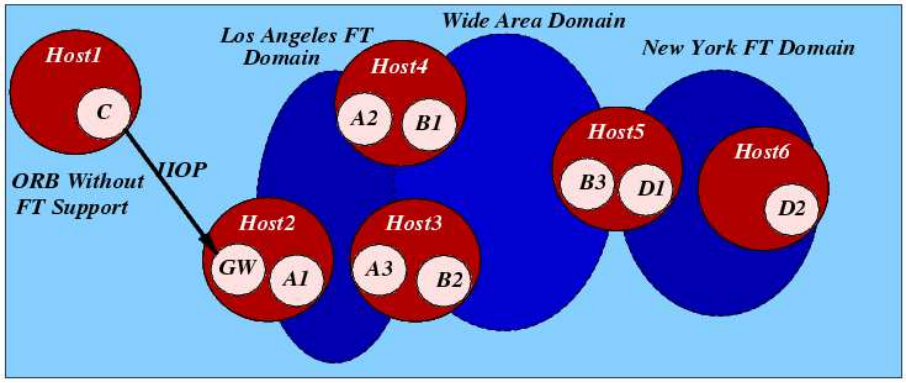

Figure 2.7: FT-CORBA domains

- **Fault Tolerance Domain (FTD)** - FT-CORBA introduces FTD (Fig. 2.7 [9]) to allow application *scalability* [15]: a FTD is a set of interconnected hosts where object groups reside; all the object groups within a FTD are managed by a single Replication Manager (*section 2.3.2*); a host may belong to one or more domains, while members of the same group may reside on different hosts. For instance, in Fig. 2.7, *Host4* belongs to the *Los Angeles FT Domain* and to the *Wide Area Domain*; the group *A* is composed by *A1*, *A2* and *A3* residing on different hosts.

- **Object Group Fault Tolerance Properties (FTP)** - FTP contain the specifications about the behaviour of an object group, concerning aspects such as:

  - *Object Replication Technique*
  - *Group Composition and Deployment*
  - *Group Consistency*
  - *Replicas fault-monitoring*

  FTP are set by the Replication Manager (*section 2.3.2*).

2.3.2 Replication Management

Replication Management consists of the creation or removal of object groups, of object group members and FTP assignement and modification; in each FTD a **Replication Manager** is present and is responsible for such activities. FT-CORBA replication styles are:

- **Cold passive replication** - clients send their requests to the primary member; when the operation has been completed, the state of this object is recorded in a log.
- **Warm passive replication** (or Primary-Backup replication) - clients send their requests to the primary member; when the operation ends, the primary member sends update messages to the backup members; they apply the changes produced by the invocation execution at the primary member; in order to maintain consistency, the communication between the primary and the backups must be reliable and updates must be processed by all the backups in the same order (*FIFO consistency*).
- **Active replication** - all the members of the object group receive and process indipendently the request sent by the client; consistency is guaranteed by the fact that replicas will produce the same output if the inputs are provided in the same order: no coordination mechanisms is necessary; a reliable multicast group communication system must be provided; the client must receive only the fastest reply: to avoid client to receive multiple replies, the ORB must pass to the client only one reply relative to a request and discard all the others.

The Replication Manager services are distributed among the following modules:

- **Property Manager** - it sets FTDs and FTP;
- **Object Group Manager** - it is able to add or remove single members of an object group;
- **Generic Factory** - it is able to create or destroy an object group.

2.3.3 Fault Management

The main role of the **Fault Management** is the detection of object failures; it can also create *fault reports* and notify their analysis; fault management consists of (Fig. 2.8):

- **Fault Detectors** - they are FTD specific; two mechanisms are available to detect failures:
  - *pull-based monitoring* (polling) - the Fault Detector invokes periodically the *is_ alive* method on each replicated object which must

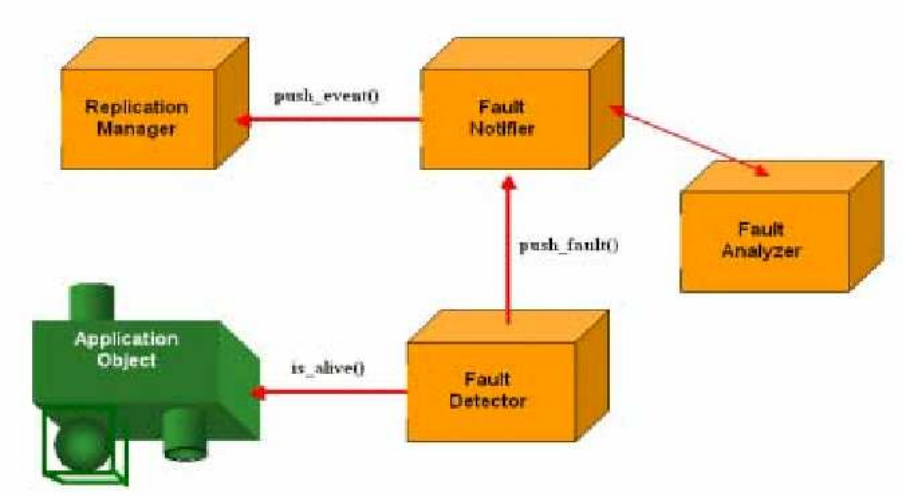

Figure 2.8: Fault detection and notification

return its *alive* state within a given time, else the Fault Detector assume that object as crashed and passes a fault report to the Fault Notifier;

- *push-based monitoring* (heartbeat) - objects periodically send to the Fault Detector a special message to say that they are alive; if the Fault Detector does not receive such a message from an object after a certain time, it considers that object as failed.

- **Fault Notifier** - it is FTD specific and forwards the received fault reports to the Replication Manager and to other subscribed clients such as the Fault Analyzer;

- **Fault Analyzer** - it receives the fault reports from the Fault Notifier and performs some kinds of analysis on it.

2.3.4 Recovery Management

Recovery Management consists of two mechanisms:

- *logging* - it stores in a log file the information about the objects such as state, incoming requests and replies after each operation has been completed (Fig. 2.9 [9]);

- *recovery* - it sets the state of a member retrieving the information from the log file (Fig. 2.10 [9]); recovery is applied when either after a fault, a backup member of an object group has been promoted to the primary member, or when a new member is introduced in the object group; the recovery processes the log messages to set the correct current state of that object. The recovery steps in the case of fault, change depending on the adopted replication logic:
  - **Cold passive replication** - if the primary fails, the new elected primary state must be updated to state of the failed primary; since using this kind of replication logic, backups do not receive any information about primary member operations, all the messages recorded in the log file are played back to the new primary; in this way the state of the new primary is consistent with the state of the old primary before the failure;
  - **Warm passive replication** - since backups receive information about primary member's activity and state, the recovery mechanism applies to the new primary only the recent state updates, those after the last complete operation;
  - **Active replication** - if an object group member fails, the requests are still processed by the other group members producing replies.

  Using passive replication, after a fault, when all the replicas are consistent, the recovery mechanism re-invokes the operation requested by the client on the new primary replica.

2.4 FT-CORBA limitations

The limitations of FT-CORBA depends on allowing developers to extend FT-CORBA specifications in their system implementations to meet their application specific reliability requirements; the main limitations are:

- *Lack of interoperability* - FT-CORBA specifies that inside a FTD, all the hosts must use the same ORB and the same fault tolerant policy;
- *Fault tolerance limited to crash failures* - in the FT-CORBA standard, only object crashes are considered as faults to be tolerant to; fault tolerant for other kinds of fault is not guaranteed, such as objects giving uncorrect results, networking faults or problems due to network partitioning;

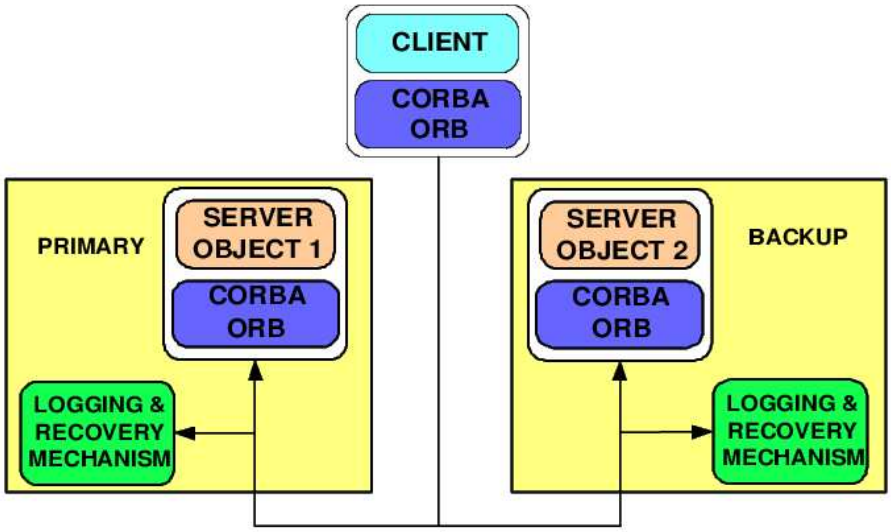

Figure 2.9: Logging process for passive replication

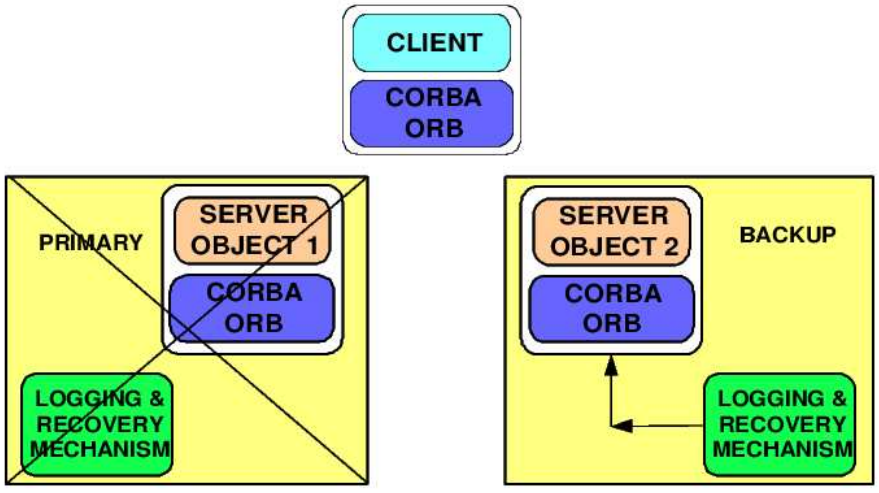

Figure 2.10: Recovery process

- *Deterministic behaviour* - object replication consistency may be compromised by non-determinism, meaning that consistency is maintained assuming that all object replicas produce exactly the same results with identical inputs.

FT-CORBA specification has been intentionally left open leaving several issues to be solved with proprietary solutions by the system developers.

Chapter 3

A real case study: DOORS

3.1 Introduction to DOORS

The *Distributed Object-Oriented Reliable Service* (**DOORS**) [9] was developed before OMG issued the FT-CORBA standard, as an experimental fault tolerant CORBA middleware; some of the concepts and the implementing solutions introduced in DOORS are present in FT-CORBA specification. Explaining the DOORS functionalities, we can understand how DOORS influenced the FT-CORBA standard.

3.2 DOORS components

Fig. 3.1 [9] shows the components of the DOORS architecture; DOORS has a **Replication Manager** providing most of the property management and group management specified in FT-CORBA (*section 2.3.2*); the **Fault Detector** and the **Super Fault Detector** provide hierarchical fault detection and notification: the Fault Detector is responsible to detect faults at object level, while the Super Fault Detector at host level. They have another role: they act as *Fault Notifiers* propagating fault reports directly to the Replication Manager.

Fault Detectors supports both *pull-based* and *push-based* monitoring (*section 2.3.3*); the Fault Detectors and the Replication Manager are monitored by a Super Fault Detector in a push-based way; since FT-CORBA does not allow single points of failures, the Super Fault Detector itself is replicated and the replicas monitor each-other by push-based monitoring; among Super Fault Detector replicas, there is a primary; if it fails, backups elect a new primary.

DOORS defines a way to dinamically set the number of missing "heart-

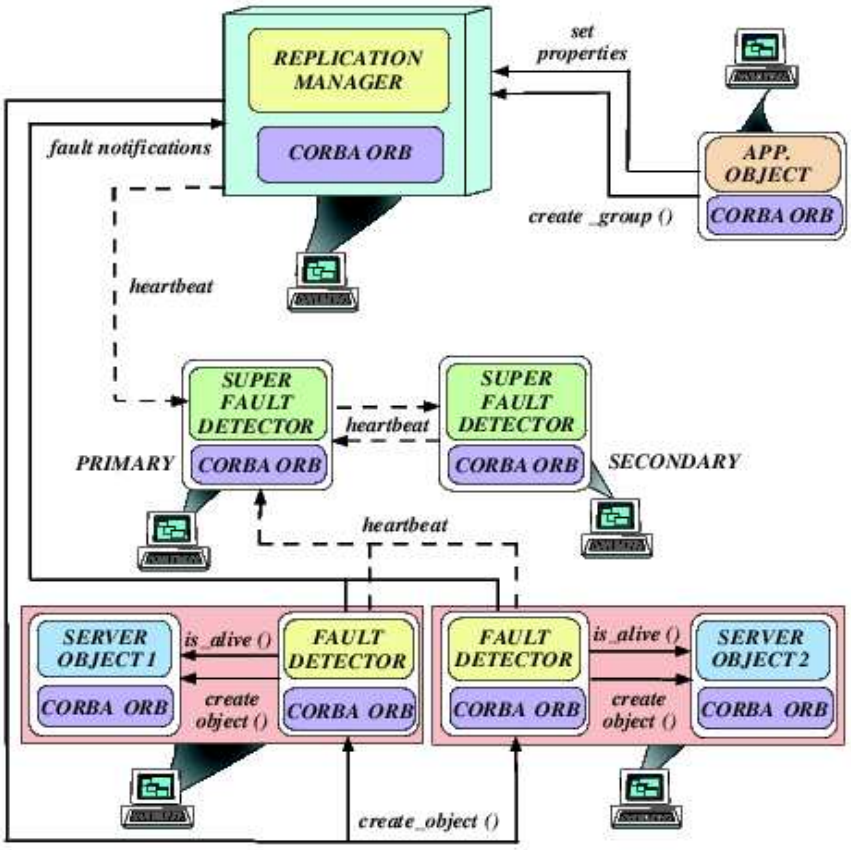

Figure 3.1: DOORS architecure

beats" necessary to consider a replica as failed using the pushed-based monitoring; this is useful to prevent unnecessary fault detection and recovery.

Fig. 3.2 [9] shows the components interaction when a replica group is established using **warm passive replication** (*section 2.3.2*):

1. an application manager requests the Replication Manager to create an object group with specific Fault Tolerant Properties (FTP) (*section 2.3.1*);

2. the Replication Manager delegates this work to its **Factory** (*section 2.3.2*) objects; they return, when the replicas are created, the corresponding object references;

3. the Replication Manager orders the Fault Detectors to monitor the new objects;

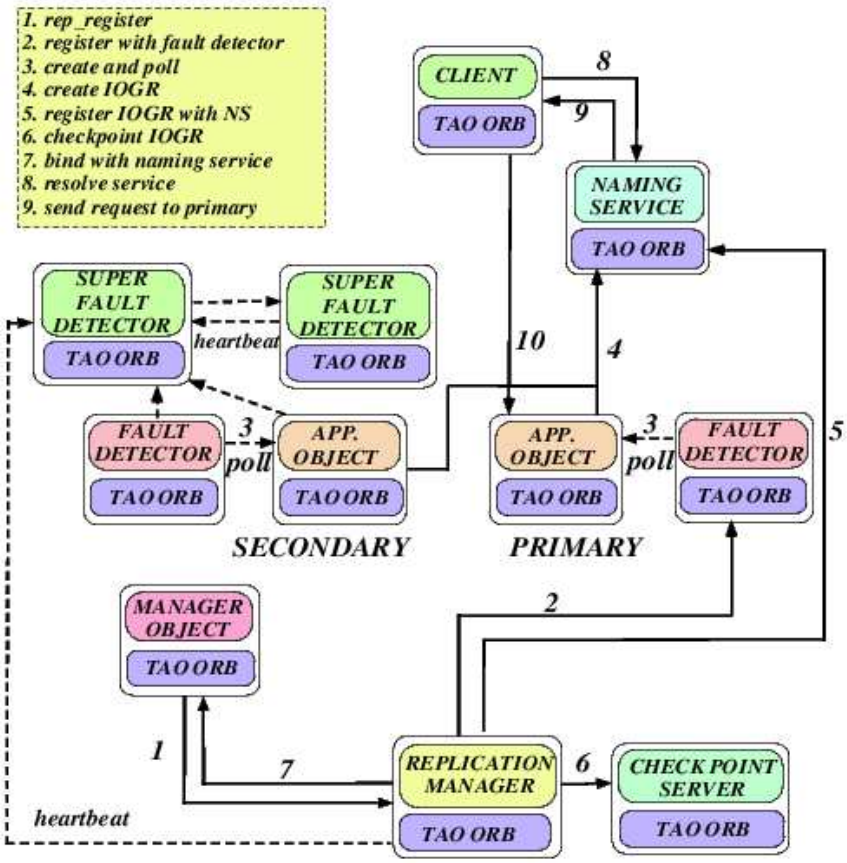

Figure 3.2: DOORS components interaction

4. the Replication Manager collects all the IORs of the new objects, generating the IOGR (*section 2.3.1*) of the object group with the indication of the primary replica;

5. the Replication Manager registers the IOGR at the **Naming Service**; in this way, the IOGR is available for other CORBA applications;

6. the Replication Manager checkpoints the IOGR;

7. a client needing the service corresponding to the new object group, contacts the Naming Service to obtain the relative IOGR;

8. the Naming Service passes to the client the relative IOGR;

9. the client make the request and the client ORB delivers the request to the primary object inside the group.

3.3 Enhanced IOGR

DOORS provides several extensions that are not specified in FT-CORBA standard; one of them is the **Enhanced IOGR**: if an object group is composed by n members, the DOORS Replication Manager creates an IOGR with $n+1$ profiles; the added profile is a reference to the Replication Manager; when an object of the group fails, the Fault Detector inform the Replication Manager about the object failure; now the IOGR needs to be updated because the group membership has changed; in FT-CORBA, the IOGR held by client is updated when the client makes a request using that IOGR; in DOORS, the client continues to use the same IOGR to invoke an operation; after all replicas in the IOGR held by the client, have been tried, the ORB uses the last reference to invoke the Replication Manager which passes to the client the new IOGR.

In this way, the client uses the same IOGR until at least one of the references is valid and obtains a new IOGR only when all the objects corresponding to the reference inside the IOGR held by the client, are failed. The overhead due to IOGR updates to the clients is significantly reduced.

3.4 Unimplemented FT-CORBA features in DOORS

These FT-CORBA features are not present in DOORS:

- **Active replication** - DOORS is able to create object group using active replication , but it does not support yet a group communication protocol for this replication style;
- **Logging and recovery** - DOORS supports application-controlled logging and recovery system, but it does not support infrustructure-controlled logging and recovery;
- **Fault Notifier** - a Fault Notifier is not present in DOORS; fault detection and notification are both provided by the Fault Detectors.

Bibliography

[1] Object Management Group. Description of New OMA Reference Model. *OMG Document*, ab/96-05-02, May 1996.

[2] D. C. Schmidt. *Overview of CORBA*. http://www.cs.wustl.edu/∼schmidt/corba-overview.html.

[3] S. Vinoski. CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 35(2), February 1997.

[4] Object Management Group. *The Common Object Request Broker: Architecture and Specifications*. http://www.omg.org.

[5] S. Vinoski. Distributed Object Computing with CORBA. *C++ Report Magazine*, 5, July/August 1993.

[6] S. Vinoski. New Features for CORBA 3.0. *Communications of the ACM*, 41(10), October 1998.

[7] Object Management Group. CORBA Messaging Joint Revisited Submission. *OMG Document*, orbos/98-05-05, 1998.

[8] Object Management Group. Object by Value. *OMG Document*, orbos/98-01-18, 1998.

[9] Balachandran Natarajan, Aniruddha S. Gokhale, Shalini Yajnik, and Douglas C. Schmidt. DOORS: Towards high-performance fault tolerant CORBA. In *Proceedings of International Symposium on Distributed Objects and Applications*, 2000.

[10] P. Narasimhan et al. L. E. Moser, P. M. Melliar-Smith. The Eternal System: an Architecture for Enterprise Applications. In *Proceedings of 3rd International Enterprise Distributed Object Computing Conference*, 1999.

[11] C. Sabnis et al. M. Cukier, J. Ren. AQuA: An Adaptive Architecture that Provides Dependable Distributed Object. In *Proceedings of IEEE 17th Symposium on Reliable Distributed Systems*, 1998.

[12] Object Management Group. Fault Tolerant CORBA specification. *OMG Document*, ptc/2000-04-04, 1999.

[13] A. Virgillito R. Baldoni, C. Marchetti and F. Zito. Failure Management for FT-CORBA Applications. In *Proceedings of the Sixth International Workshop on Object-Oriented Real-Time Dependable Systems*, 2001.

[14] R. Baldoni C. Marchetti, M. Mecella. Architectural Issues on Fault Tolerance in CORBA. In *Proceeding of the SSGRR Computer and Business Conference*, 2000.

[15] M. Van Steen A. S. Tanenbaum. *Distributed Systems: Principles and Paradigm*. Prentice Hall, 2002.