

# The Fault Tolerant CORBA Architecture

Estiévenart Fabrice

Facultés Universitaires Notre-Dame de la Paix, 5000 Namur - Belgium

**Résumé** In this paper, we present the architecture of the Fault Tolerant CORBA as defined in the OMG specification v2.6. We see how object groups can be referenced with the use of IOGRs, how failure transparency can be achieved and what are the replication styles proposed by the OMG. Finally we explain how faults can be detected and once they are, how they are managed by the architecture.

## 1 Introduction

A distributed application is said fault-tolerant if it can be properly executed despite the occurrence of faults. Many different classes of distributed applications need high reliability and therefore may require fault-tolerance : air traffic control systems, e-commerce applications, WEB services, telecommunication systems, etc. When we talk about reliability of distributed systems two contradicting facts become apparent. Since distributed systems are made up of many components, there is no single point of failure. But since there are more number of components, the probability of failure of any single component also increases.

Generally, fault-tolerance can be obtained by software redundancy : a service can survive to a fault if it is provided by a set of server replicas. If some replicas fail, the others can continue to offer the service.

CORBA systems have long lacked real support for fault tolerance. In most cases, a failure was simply reported to the client and the system undertook no further action. For example, if a referenced object could not be reached because its associated server was unavailable, a client was left on its own. In CORBA version 2.6, fault tolerance is explicitly addressed.

This paper gives a short presentation and overview of the current Fault Tolerant CORBA (FT-CORBA) specification. Section 2.1 introduces the main ideas inherent in the concept of fault tolerance. Section 2.2 presents a global overview of the FT-CORBA architecture. In the FT-CORBA specification, fault-tolerance is achieved through object replication, fault detection and fault recovery. The three next sections give details on how these mechanisms should be implemented. Finally, we will see which limitations the current FT-CORBA specification suffers from.

## 2 Fault Tolerant CORBA

### 2.1 Basic Concepts

**Replication transparency** The basic approach for dealing with failures in CORBA is to replicate objects into object groups. Such a group consists of one or more identical copies of the same object implementing the same interface. A group offers the same interface as the replicas it contains. However, an object group is created, managed and accessed by clients as a single entity. In other words, replication is transparent for the client[LM] :

- the client application code used for binding to an object group is the same as for binding to a single object.
- the client sends a single request and receives a single reply - as it would with a single object - even if multiple servers reply to the client
- the reliability protocols handled by the object group are invisible to the client

Because of this flexibility, fault tolerance may be added to the system even after client applications have already been deployed. This is done by simply replacing the non-fault tolerant server objects with their fault tolerant counterparts.

**Fault Tolerance Domains[OMG01]** Many applications that need fault tolerance are quite large and complex. Managing such an application as a single entity is inappropriate. Consequently, the OMG specification for FT-CORBA defines fault tolerance domains. Each fault tolerance domain typically contains several hosts and many object groups. All of the objects groups within a fault tolerance domain are created and managed by a single Replication Manager. The concept of fault tolerance domains allows applications to scale to arbitrary sizes, by allowing a smaller number of objects to be managed by each Replication Manager.

**Interoperable object group references (IOGRs)[BMPV]** To provide replication transparency as much as possible, object groups should not be distinguishable from normal CORBA objects. An important issue, in this respect, is how object groups are referenced. The approach followed is to use a special kind of multiprofile IOR, called Interoperable Object Group References (IOGR). Each profile contained in the IOGR identifies a connection end-point and holds the information needed by the ORB to reach a CORBA object implementation. Moreover, an IOGR is able to handle membership changes within an object group, object group versioning and primary identification within an object group. Fig.1 shows an example structure of a generic IOGR. Each IOP-profile contains one TAG\_GROUP component. This component is a struct composed by four fields; these fields allow to uniquely identify an object group in the context of a fault tolerance domain and to associate version numbers to membership changes.

- `tag_group_version` : the version of this tagged component.
- `ft_domain_id` : the identifier of a Fault Tolerance Domain this IOGR belongs to.

- `object_group_id` : the identifier of an object group. This identifier is assigned when an object group gets created.
- `object_group_version` : the version of this object group reference. This information can be used to verify the validity (freshness) of the reference and if there is need to update the IOGR that the client is using <sup>1</sup>.

The `TAG_FT_PRIMARY` component of a IIOP profile, if present, informs to the client ORB that the containing IIOP profile probably is the primary member of a passively replicated object group <sup>2</sup>. At most one of the IIOP-profiles (or fault tolerance group components) can be marked as a primary object group member with this `TAG_FT_PRIMARY` component.

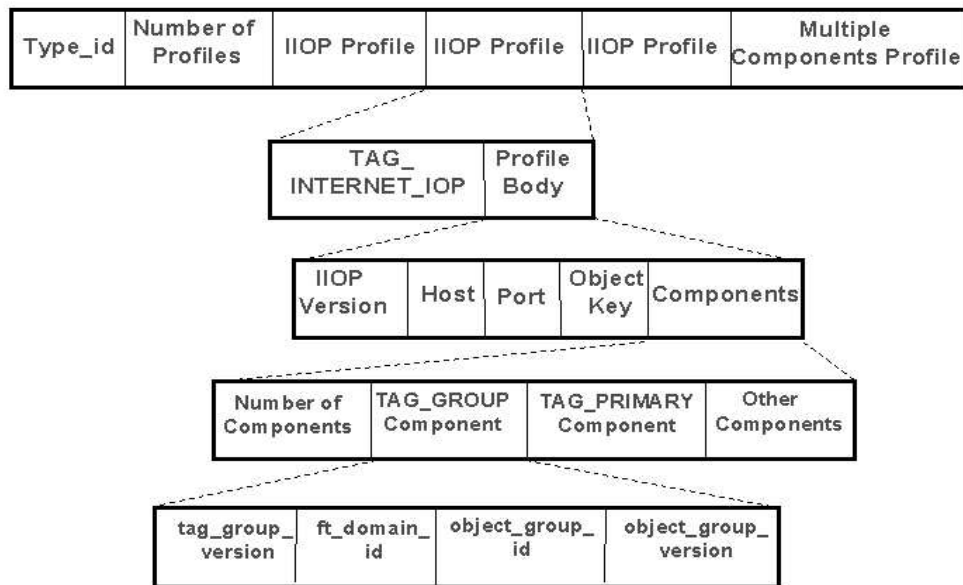


FIG. 1. The structure of an IOGR

**Failure transparency** A client of a replicated object has to be masked as much as possible from failures of object group members or of the network. To deal with such issue, in the FT-CORBA standard two mechanisms are identified, i.e. transparent client reinvocation and transparent client redirection[BMPV].

*Transparent Client Reinvocation* When a client wants to invoke a method on a remote object group, it passes an IOGR to its ORB which then tries to bind

<sup>1</sup> see next section

<sup>2</sup> Passive Replication is detailed in 2.3

one of the referenced profile contained in the IOGR. If this attempt fails, the FT-CORBA redirection mechanism allows to select automatically another profile from the IOGR and try again the connection. The transparent client reinvocation mechanism provides clients with replication and failure transparency exploiting two sub-mechanisms that we define *transparent client failover* and *transparent request identification*. The transparent client failover mechanism mandates a FT-CORBA compliant client ORB, invoking an object group identified by an IOGR, to try exploiting all of the profiles contained in the object reference before returning an exception to the client. More precisely, a FT-CORBA compliant client ORB must not abandon an invocation (throwing an exception to the client) until :

- it has tried to invoke the replicated servers by exploiting all of the profiles contained in the IOGR while receiving only exceptions falling into the so called failover conditions <sup>3</sup>
- it has received an exception that does not fall among such failover conditions
- the request expiration time has elapsed
- it has received the reply.

To preserve the CORBA 'at most once' request semantic, additional information is included by any FT-CORBA compliant client ORB in each request so that repeated executions upon client reinvocations are avoided. FT-CORBA defines how a client ORB must uniquely identify a request to allow servers to automatically perform duplicate filtering, and this is what we have defined as *transparent request identification*. Request identification is done in FT-CORBA by using the REQUEST service context. This service context is put in each request and contains three fields, namely : `client id`, `retention id` and `expiration time`. The pair composed by `client id` and `retention id` is a unique request identifier that allows the server ORB to identify repetitions. If the request is a repetition, the server does not reexecute the request but rather returns the reply that was generated by the prior execution. The `expiration time` provides to servers a lower bound on the time until which they must retain the request information and the corresponding reply.

*Transparent Client Redirection* IOGRs held by clients may become obsolete : when a group member fails or a new member joins a group, a membership change can occur asynchronously with respect to the client invocations. In this case, the IOGR hold by the client to contact the group can't be dynamically updated. For this reason, FT-CORBA defines how to propagate membership changes without having the client resolve the object group reference each time it has to perform an invocation. The solution to this problem adopted by FT-CORBA is based on the assumptions that object group members rapidly track membership changes and

---

<sup>3</sup> Failover conditions are the circumstances under which an ORB is allowed to reissue a request using a (possibly different) profile contained in an IOGR

always have updated references <sup>4</sup>. Thus, to provide clients with the most updated IOGRs, a server needs to know which is the current version of the IOGR held by the client. For this reason, a FT\_GROUP\_VERSION service context is defined. It contains a single field `object_group_ref_version` in which the client ORB copies the `object_group_ref_version` field value of the IOGR's TAG\_GROUP component. In this way, when a group member server ORB receives a request, it can determine whether the client reference is obsolete or not. In the first case, it throws an exception to the client ORB that, when received by the client ORB, has the effect of permanently updating its reference. In the second case, the request is normally executed and a reply is returned to the client.

## 2.2 Architectural Overview[OMG01]

Fig.2 presents an architectural overview of a fault-tolerant system, showing an example strategy for implementation of the specifications for FT-CORBA. Other implementation strategies are possible. A client object C invokes a method in a transparent way on an object group composed of two server replicas S1 and S2, located on two different hosts H2 and H3.

Replicated objects are managed by the Replication Manager which inherits three interfaces :

- the Object Group Manager which is called by the application for managing object groups.
- the Generic Factory which calls local factories located on each hosts for managing group members.
- the Property Manager which defines the properties of a group such as the replication or the membership style.

When a fault occurs, it is detected by Fault Detector objects which are located on each host server and which can be supervised by an additional Fault Detector object. When a fault is detected, it is transmitted by Fault Notifier to fault consumers, such as the Replication Manager. Each server owns a logging and recovery mechanism to recover from faults.

## 2.3 Object Replication

The Replication Manager inherits three application program interfaces : PropertyManager, ObjectGroupManager, and GenericFactory.

**Property Manager** The Property Manager is used to set properties of object groups. Such a property is the Replication Strategy.

---

<sup>4</sup> membership changes can be managed in a centralised way by a group server or in a distributed way via 'Hello' and 'Goodbye' messages transmitted between group members[TvS02]

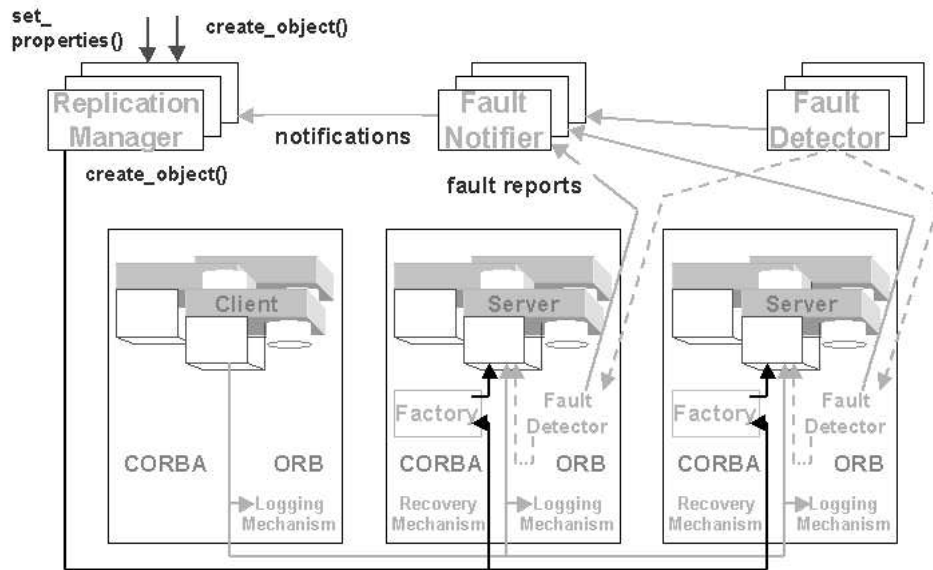


FIG. 2. The FT-CORBA Architecture

*Replication Strategy* The FT-CORBA has five replication styles, which define how object group members are replicated within object group. Replication styles affect the way the state of an object gets recorded, when the state recording takes place, and how the fault recovery can be done.

*Stateless* : The object contains read only data. Object's state can't be modified, so there is no need for recording or transferring it.

*Cold Passive replication* : In a Cold Passive replication strategy, clients send their requests to a unique primary member of the group which is the only one who executes the requests. After the operation has completed, the state of this primary gets recorded in a message log. When faults occur, the recovery is done using the state information and messages recorded in the message log. The state gets transferred to a new primary member object. This kind of a recovery is the slowest recovery mechanism in the FT-CORBA.

*Warm Passive replication* : The basic principle of Warm Passive replication, also called Primary Backup replication [TvS02], is that clients send their requests to a primary, which executes the requests and sends update messages to the backups. The backups do not execute the invocation, but apply the changes produced by the invocation execution at the primary. Contrary to Active Replication, no determinism constraint is necessary on the execution of invocations. In order to maintain consistency within the group, communication between the primary and the backups has to be reliable and to guarantee that updates are processed in the same order on all the backups. This kind of consistency is cal-

led FIFO consistency[TvS02]. However, primary backup communication based on FIFO channels is not enough to ensure correct execution in case of failure of the primary. For example, consider that the primary fails before all backups receive the updates for a certain request, and another replica takes over as a new primary. Some mechanism has to ensure that updates sent by the new primary will be 'properly' ordered with regard to the updates sent just before by the faulty primary. Warm Passive replication provides faster recovery from faults than Cold Passive replication and uses little processing power when compared to other replication techniques. However, Warm Passive replication suffers from a high reconfiguration cost when the primary fails.

**Active Replication :** Active replication is a non-centralised replication technique. Its key concept is that all members of the object group receive and simultaneously process the same sequence of client requests in an independent way. Consistency is guaranteed by assuming that, when provided with the same input in the same order, replicas will produce the same output. This assumption implies that servers process requests in a deterministic way. Therefore, no expensive coordination mechanisms are necessary to keep up consistency. Clients do not contact one particular server, but address servers as a group. In order for servers to receive the same input in the same order, the fault tolerance infrastructure must provide multicast group communication system that has reliable totally-ordered message delivery (atomic multicast[TvS02]). The main advantage of active replication is its simplicity (e.g., same code everywhere, no need for consistency protocol) and failure transparency. Failures are fully hidden from the clients, since if a replica fails, the requests are still processed by the other replicas. This type of recovery mechanism provides very rapid recovery from faults. The determinism constraint is the major drawback of this approach. Although one might also argue that having all the processing done on all replicas consumes too much resources. So, active replication can be useful when the cost of transferring a state is larger than the cost of executing a method invocation, or when the time available for recovery after a fault is tightly constrained. Note that to be consistent with the replication transparency principle, only the fastest reply is received by the client application.

**Active replication with Voting :** Also called quorum-based replication[TvS02], this replication style is quite similar to the Active replication. Replies from the members of the source object group are voted, and are delivered to the members of the destination object group only if a majority of the replies are identical (match exactly). This Replication Style is not supported in the current specification, but is an anticipated extension. Voting itself is computationally inexpensive but the communications infrastructure required to support voting properly is substantially expensive[CDM]. Quorum-based strategies can be found in [SO97]

**Object Group Manager** The ObjectGroupManager interface provides operations that allow an application to exercise control over the creation, addition, removal and locations of members of an object group. When creating and adding a new member in an object group, the application is allowed to choose

the physical location at which the new member is to be created and the object group reference to which the member is to be added. This interface also provides methods to obtain the current reference and identifier for an object group.

**Generic Factory** The `GenericFactory` interface is generic in that it allows the creation and deletion of replicated fault tolerant application object (object group), individual members of an object group and unreplicated objects. The client remains unaware of the fact that it is implicitly creating an object group. Using the `create_object()` operation of the `GenericFactory` interface, the application program requests the creation of an object group, just as it would an unreplicated object. This operation is actually invoked on the `Replication Manager`, rather than directly on the factory (as it would have been in the unreplicated case). To create local members of a group or local unreplicated objects, the `Replication Manager` then invokes the factories, on the different hosts, using the same `create_object()` operation of the `GenericFactory` interface. The number of replicas that are created when starting a new object group is determined by the `InitialNumberReplicas` value. The `Replication Manager` is also responsible for replacing a replica in the case of a failure, thereby ensuring that the number of replicas does not drop below the specified `MinimumNumberReplicas` system-dependent value.

## 2.4 Fault Management

In a fault-tolerant system, fault management encompasses the following activities :

- Fault detection : detecting the presence of a fault in the system and generating a fault report.
- Fault notification : propagating fault reports to entities that have registered for such notifications.
- Fault analysis/diagnosis : analyzing a (potentially large) number of related fault reports and generating condensed or summary reports.

In the `Fault Tolerance Infrastructure`, `Fault Detectors` detect faults in the objects, and report faults to the `Fault Notifier`. The `Fault Notifier` receives fault reports from the `Fault Detectors`, filters the reports, and propagates the filtered reports as fault event notifications to consumers that have subscribed for them. The `Fault Analyzer` reasons about the fault reports that it has received, and produces aggregate or summary fault reports that it propagates back to the `Fault Notifier` for dissemination to other consumers. In the following sections, we examine in details how these fault management objects work.

**Fault Detector** Fault detection is the timely ability to identify a fault's existence and location. Principally in a distributed system, server objects and network faults must be detected [tSs]. A fault-tolerant system typically has several `Fault Detectors`. Each `Fault Detector` belongs to a particular fault tolerance domain, and is not shared across fault tolerance domains. `Fault Detectors` can

be located at several levels in hierarchy. A Fault Detector can monitor one or more objects within a process, or monitor objects of several processes within a host. There are two common styles of fault monitoring : PULL-based and PUSH-based. These two fault monitoring styles differ in the direction in which fault information flows in the system. In a push-based approach, also referred to as server-based protocols [TvS02], faults are reported to the Fault Detector by faulty objects. In the pull-based approach, also called client-based protocols [TvS02], the Fault Detector periodically pings the member by invoking an `is_alive()` operation. If no answer is received by the Fault Detector within a specified time-out, server object is considered faulty. The fault management CORBA specification defines the interaction between a *pull-based Fault Detector* and application objects. It defines a `PullMonitorable` interface that the application objects inherit. The period of the ping is determined by the `FaultMonitoringInterval` for the object group. Fault Detectors, from whatever source, push fault reports onto Fault Notifier channels. The Fault Detectors and Fault Notifier use a well-defined event type to convey a given fault event. The OMG specification defines a set of fault event types that are understood by the Fault Tolerance Infrastructure.

**Fault Notifier** The Fault Notifier works as a publisher who disseminate fault event to event subscriber. Any fault event supplier (Fault Detector) may obtain the reference to the Fault Notifier and send fault reports to it. The Fault Notifier then performs report filtering based on constraints provided by the consumer in order to eliminate unnecessary or duplicate reports. It finally sends fault event notifications to consumers that have registered for such information. Such consumers are the Replication Manager, the Fault Analyzer <sup>5</sup> or other application objects. Logically, there is one Fault Notifier per fault tolerance domain but in real fault tolerance systems Fault Notifiers are also replicated to provide better reliability.

**Fault Analyzer** A problem with fault notification is the potential for a large number of notifications to be generated by a single fault. This problem is addressed by Fault Analyzers which work as consumers of fault reports generated by the Fault Notifier. They perform event correlation, analysis, and diagnosis in order to generate a filtered condensed fault report that they send to fault event consumers.

## 2.5 Recovery Management

So far, we have explained how a FT-CORBA architecture is able to detect and to notify faults. However, once a failure has occurred and has been detected, it is essential that the process where the failure happened can recover to a correct and consistent state.

---

<sup>5</sup> see next section

The fault tolerance system includes a backward recovery and a logging mechanism. In backward recovery, the main issue is to bring the system from its present erroneous state into a previously correct state. To do so, it is necessary to record the system's state from time to time, and to restore such a recorded state when things go wrong. Each time (part of) the system's present state is recorded, a checkpoint is said to be made. Restoring a single faulty object to consistent state (object rollback) can be one of the simplest fault tolerant services. It's as simple as instantiating a new object with a state consistent with the last checkpoint. Recovering a system composed of multiple distributed objects by which each object simply records its local state from time to time in an uncoordinated fashion is often more difficult. If local object states jointly do not form a consistent global state, further rolling back is necessary and this process of a cascaded rollback may lead to what is called the domino effect[TvS02]. Distributed checkpointing requires that different objects coordinate checkpointing to achieve global consistency but coordination requires global synchronization, which may introduce performance problems. Considering that checkpointing is a costly operation, techniques have been sought to reduce the number of checkpoints, but still enable recovery. An important technique in distributed systems is logging messages. The basic idea underlying message logging is that if the transmission of messages (requests or replies) can be replayed, we can still reach a globally consistent state but without having to restore that state from safe stable storage[TvS02]. The FT-CORBA system combines checkpointing (states and updates recording) with message logging (request and reply messages recording) but in the OMG specification, these two techniques are merged into the unique abstract term 'logging'.

**The Logging mechanism** During normal operation, the Logging Mechanism records the state and actions of the primary member of a passively replicated object group in a log. The log may be distributed, in which case it is maintained in local volatile storage at each member of the object group. Alternatively, as it is the case for the passive replication strategy, the log may be written to shared stable storage by the primary member of the object group before it is transmitted to all of the members of the object group. The format of the log is not specified in this specification. Typically, the information recorded in the log consists of states, updates, request and reply messages. The log must preserve the order in which messages were received by the members of the object group, so that they can be replayed in the correct order during recovery.

**The Recovery mechanism** The Recovery Mechanism sets the state of a member, either after a fault, when a backup member of an object group is promoted to the primary member, or alternatively when a new member is introduced into an object group. The Recovery Mechanism processes the log and applies messages from the log to the member to bring that member to the correct current state, so that it can start to process messages normally. The messages in the log are not necessarily in the order required for recovery. The Recovery Mecha-

nism processes the log, discarding irrelevant messages to form a complete log. A complete log for an object group contains :

- The most recent complete state in the log.
- All complete updates that occur after the most recent complete state.
- All request and reply messages that occur in the log after the most recent complete state and after the most recent complete update, if present.

The entire complete log must be applied to an object in two cases :

- a backup member of an object group with the Cold Passive Replication strategy that is being promoted to primary member.
- a new member of an object group with the Active Replication strategy

In the case of a Warm Passive Replication strategy, a backup member of an object group that is being promoted to primary member, has already received states and updates during normal operation. In order to speed up the recovery process, the Recovery Mechanism applies to the member, only messages in the complete log that follow the most recent state or update applied to the member during normal operation.

An application object inherits the Checkpointable interface, which provides `get_state()` and `set_state()` operations, to enable the Logging and Recovery Mechanisms to record and restore its state. The interval between successive invocations of the `get_state()` operation is determined by the `CheckpointInterval` value from the Property Manager. In order to reduce the log size, objects may also inherit the Updateable interface, which provides `get_update()` and `set_update()` operations. An update is a data that represents the change (delta) between the previous state (the state at the moment of the most recent invocation of `get_state()` or `get_update()`) and the current state.

## 2.6 FT-CORBA Limitations

The FT-CORBA specification has many compromises and several issues have been intentionally left open. The limitations of the FT-CORBA specification are given below.

**Legacy ORBs** A client, running on legacy ORB, may invoke methods from fault tolerant servers and receive fault-tolerant service without any knowledge of fault-tolerance.

**Common Infrastructure** In order the fault tolerance to work properly it is required that all replicas of an object must be hosted by the infrastructure from the same ORB vendor.

**Deterministic Behavior** In case of Active Replication, strong replication consistency can uniquely be maintained if all object replicas produce exactly same results with identical inputs.

**Network Partitioning Faults** Network partitioning faults separate the hosts of the system into two or more sets, each set being unable to operate and to communicate with hosts of different sets. The specification does not solve networking faults and problems caused by network partitioning.

**Commission Faults** The specification does not try to solve or protect against cases where objects or hosts generate incorrect results, maliciously.

**Correlated Faults** The specification does not provide protection against software design or programming faults.

### 3 Conclusion

The FT-CORBA specification adds fault tolerance features to standard CORBA with minimal modification to existing ORBs. Fault tolerance is mainly achieved by object replication. Replicated objects are gathered into object groups that are referenced by a multiple IORs structure called IOGR. The specification also adds two new service contexts to manage failure transparency : the REQUEST service context and the FT\_GROUP\_VERSION service context. Within a fault tolerance domain, the FT-CORBA supports both passive and active replication, with some variants. Fault management consists in detecting, notifying and recovering faults so that the entire system constantly seems consistent for the client application. The specification has been intentionally left open. It still has limitation and leaves several issues for vendors to solve with proprietary solutions.

## Références

- [BMPV] R. Baldoni, C. Marchetti, R. Panella, and L. Verde. Handling ft-corba compliant interoperable object group references. Dipartimento di Informatica e Sistemistica, Università a di Roma 'La Sapienza', Via Salaria 113, 00198, Roma, Italy.
- [CDM] G. Chockler, D. Dolev, and D. Malkhi. A quorum based approach to corba fault-tolerance. School of Computer Science and Engineering, The Hebrew University of Jerusalem, Jerusalem 91904, Israel.
- [LM] S. Landis and S. Maffeis. Building reliable distributed systems with corba. Isis Distributed Systems Inc., Ithaca, NY 14850 and Dept. of Computer Science, Cornell University.
- [OMG01] OMG. The common object request broker : Architecture and specification v2.6. 25 :1-105, December 2001.
- [SO97] A.L. Sousa and R. Oliveira. A corba object replication service. April 1997. Departamento de Informatica, Universidade do Minho, Braga - Portugal.
- [tSs] the Semaphore staff. Object level fault tolerance for corba-based distributed computing.
- [TvS02] A.S. Tanenbaum and M. van Steen. *Distributed Systems : Principles and Paradigms*. Prentice Hall, 1st edition, January 2002.