Date: June 2011

Lightweight Fault Tolerance for Distributed RT Systems

Version 1.0 Beta 2

OMG Document Number: ptc/2011-06-05

Standard document URL: http://www.omg.org/spec/LWFT/1.0/PDF

Associated File(s)*: http://www.omg.org/spec/LWFT/20110508

Source document: Lightweight Fault Tolerance for Distributed RT Systems Version 1.0 Beta 1 (ptc/2009-10-08)

* Original file(s): Lightweight Fault Tolerance for Distributed RT Systems machine-readable files (ptc/2011-05-08)

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.)

Table of Contents

Preface

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at http://www.omg.org/.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A Specifications Catalog is available from the OMG website at:
http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

Business Modeling Specifications

- Business Strategy, Business Rules and Business Process Management Specifications

Middleware Specifications

- CORBA/IIOP Specifications
- Minimum CORBA
- CORBA Component Model (CCM) Specification
- Data Distribution Service (DDS) Specifications

Specialized CORBA Specifications

- Includes CORBA/e and Realtime and Embedded Systems

Language Mappings

- IDL / Language Mapping Specifications
- Other Language Mapping Specifications

Modeling and Metadata Specifications

- UML®, MOF, XMI, and CWM Specifications
- UML Profiles

Modernization Specifications

- KDM

Platform Independent Model (PIM), Platform Specific Model (PSM) and Interface Specifications

- OMG Domain Specifications
- CORBAservices Specifications
- CORBAfacilities Specifications
- OMG Embedded Intelligence Specifications
- OMG Security Specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Arial - 9 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

Arial – 9 pt: Examples

NOTE: Terms that appear in *italics* are defined in the glossary. *Italic* text also represents the name of a document, specification, or other publication.

1 Scope

This specification extends the Fault Tolerance CORBA specification with capabilities to support real-time applications by providing predictable recovery times and support for application-defined consistency management.

Admitting that there is ‘no one size fits all’ solution for building highly fault tolerant solutions, applications can now provide their own fault detectors, fault analyzers and recovery mechanisms. Application state consistency will be decided and managed by application-supplied mechanisms that shall be integrated with FT infrastructures. The FT CORBA specification provided no means for applications to provide their own consistency management or to cooperate with middleware ReplicationManagement in order to synthesize a workable status for server groups.

Extensions to CORBA, FT CORBA are specified to enable interoperability between client ORBs and server ORBs in a fault tolerant enabled infrastructure.

2 Conformance

This specification defines 3 conformance levels:

1. **Server Replica Management:** Support for both cold-passive and warn-passive replication styles regardless of the interaction style (C/S, Pub/Sub).
2. **Basic Object Group Management:** Server Replica Management with support for client/server CORBA interactions based on standard IORs.
3. **Extended Object Group Management:** Basic Object Group Management with support for PortableGroups, FTCORBA IOGR's for enforcing at most once semantics, and support for Fault Detection and Notification.

An implementation claiming compliance to a given level shall also comply with lower levels (if any).

Table 1 provides a detailed service to conformance level matrix.

Services/Interfaces	**Server Replica Management**	**Basic Object Group Management**	**Extended Object Group Management**
ServerManager	X	X	X
ServerRecoveryManager	X	X	X
RecoveryObserver	X	X	X

-ORBFTLocation	X	X	X
Object Group Name		X	X
Object Group Name Alias			X
ClientFailoverSemanticsPolicy		X	X
ObjectGroupMembershipPolicy		X	X
ObjectGroupNameResolutionPolicy			X
ServiceLocator		X	X
ForwarServicedingLocator		X	X
Interoperable Object Group References			X
FT_REQUEST ServiceContext			X
FT_GROUP_ID ServiceContext			X
FTRequestDurationPolicy			X
FT Current			X
ServerCallback		X	X
ServerUpdateObserver		X	X
ServerGroupManager		X	X
ServerGroupManagerExt		X	X
RecoverableServer		X	X
Transport Heartbeats			X
FaultNotifier			X
ServerCrashFault			X
PullMonitorable			X
PullMonitorableServer			X
TCPProtocolProperties			If RT ORB

-ORBKeepAlive		X	X
GenericFactory			X
ObjectGroupManager			X
Group Object Adapter			X

Table 1: Service to Conformance level matrix

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

- [**CORBA**] Common Object Request Broker Architecture: Core Specification, March 2004 Version 3.0.3 (formal/04-03-01)
- [**FTCORBA**] Fault Tolerant CORBA - Chapter 23 from the CORBA Core Specification
- [**CCM**] CORBA Components, June 2002, Version 3.0 (formal/02-06-65)
- [**DDS**] Data Distribution Service for Real-Time Systems Specification, December 2005, Version 1.1 (formal/05-12-04)
- [**RTCORBA**] Real Time - CORBA Specification, November 2003, Version 2.0 (formal/03-11-01)
- [**UMLPROFILE**] UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, Version 1.0, (formal/06-05-02)
- [**DATAPARALLEL**] Data Parallel CORBA Specification,Version 1.0 (formal/06-01-03)
- **[MIOP]** Unreliable Multicast Inter-ORB Protocol Specification

4 Terms and Definitions

Fault Tolerance Domain

For scalability, large applications are divided into multiple fault tolerance domains each managed by a single ReplicationManager. All replicas of process groups and all the members of objects groups hosted by these processes are located within a single fault tolerance domain but can invoke, or can be invoked by, objects of other fault tolerance domains. A host can support objects from multiple fault tolerance domains.

FT-Enabled Middleware

FT-enabled middleware is responsible for providing the group abstraction needed for a reliable architecture through middleware Replication Management and transparent failover.

The FT middleware may also rely on some middleware fault detectors for monitoring the middleware infrastructure.

FT Infrastructure

The set of hardware and software components and services provided by the FT-enabled middleware and external mechanisms in support for applications in a reliable and fault tolerant system. These include replication management, fault management, consistency management and recovery management.

Process Group

A process group is a grouping of FT Process replicas managed by the ReplicationManager. Process groups host members of object groups in order to ensure replication and failure transparencies to object group users.

FT Process

A node-level POSIX process part of a process group (also referred to as replica) that is considered as a unit of redundancy by the ReplicationManager.

Object Group Manager

The Object Group Manager is responsible for handling object group membership operation requests from applications. It contains operations for creating a member of an object group at a particular location, adding a member to an object group at a particular location, removing a member from an object group at a particular location, getting the locations of the members of an object group.

Forwarder

A component of the FT-enabled middleware responsible for locating the latest object group reference and ‘forwarding’ it to the client. In order to act as a fallback object when all known object group members are not reachable by a client, an object group reference must contain a profile to a Forwarder.

Object Group Reference

In order to provide replication transparency (client objects not aware that there are several server objects) and failure transparency (client objects not aware of faults in the FT Process replicas), server object replicas are managed as an object group.

An object group reference is an object reference for the object group as a whole.

An object group reference is standard Interoperable Object Group reference (IOR) that may or may not be an Interoperable Object Group reference (IOGR).

Server

"A server is a computational context in which the implementation of an object exists. Generally, a server corresponds to a process." In LWFT a Server will always correspond to a Process

Server ID

"A Server ID must uniquely identify a server to an IMR. This specification only requires unique identification using a string of some kind and does not intend to make more specific requirements for the structure of a server ID. The server ID may be specified by an ORB_init argument of the form –ORBServerId. The value assigned to this property is a string. All templates created in this ORB will return this server ID in the server_id attribute. It is required that all ORBs in the same server share the same server ID. Specific environments may choose to implement -ORBServerId in ways that automatically enforce this requirement."

5 Symbols (and abbreviated terms)

AMSM	Application Management and System Monitoring for CMS Systems
CCM	CORBA Component Model
CIF	Component Implementation Framework
CIM	Common Information Model; a standard for system administration developed by DMTF
CORBA	Common Object Request Broker Architecture
DCE ESIOP	Environment-Specific Inter-ORB Protocol (ESIOP) for the OSF DCE environment
DCE-CIOP	DCE Common Inter-ORB Protocol
DCPS	Data-Centric Publish-Subscribe (part of DDS)
DDS	Data Distribution Service
DLRL	Data Local Reconstruction Layer (part of DDS)
DMTF	Distributed Management Task Force (cf. www.dmtf.org)
GIOP	General Inter-ORB Protocol
GOA	Group Object Adapter
IDL	Interface Definition Language

IEEE	Institute of Electrical and Electronics Engineers
IMR	Implementation Repository
IOGR	Interoperable Object Group reference
IOR	Interoperable Object Reference
MIOP	Unreliable Multicast Inter-ORB Protocol
OMG	Object Management Group (cf. www.omg.org)
ORB	Object Request Broker
OSF	Open Software Foundation
PIM	Platform Independent Model
POA	Portable Object Adapter
POSIX	Portable Operating System Interface for Unix; name of a family of related standards specified by the IEEE
PSM	Platform Specific Model
QoS	Quality of Service
RFP	Request For Proposal
RO-MIOP	Reliable Ordered Multicast Inter-ORB Protocol
UML	Unified Modelling Language
W3C	World Wide Web Consortium (cf. www.w3c.org)
WBEM	Web-Based Enterprise Management
XML	eXtensible Mark-up Language

6 Additional Information

6.1 Changes to Adopted OMG Specifications

6.1.1 Extensions

This specification adds the following to CORBA 3.1 specification:

- A new policy object: ClientFailoverSemanticsPolicy (§9.4)
- Two new POA Policies: ObjectGroupMembershipPolicy (§9.5), and ObjectGroupNameResolutionPolicy (§9.6)
- One Current object : FTCurrent (§9.25)
- Five new ObjectIds for resove_initial_references: “FTCurrent” (see §9.25), “FTServiceLocator” (§9.35), “FTObjectGroupManager” (§9.31), “FTServerGroupManager” (§9.19), and “FTServerManager” (§9.24).
- Two new ORB Configuration parameters: -ORBKeepAlive (§9.34), and -ORBFTLocation (§9.24)
- One pseudo operation: FT_Locate (§9.36).

6.1.2 Changes

This specification adds the following to FT CORBA specification:

- Add support for location agents in Interoperable Object Group References (§9.10.1)
- A new service context: FT_GROUP_ID (§9.10.2)

6.2 Acknowledgments

The following companies submitted this specification:

- Thales
- PrismTech Group Ltd

7 Overall design rationale

The proposed design overcomes shortcomings and complexities of fault tolerant CORBA by introducing the Server Group concept (generally a POSIX process group) making a Server a unit of redundancy managed by an FT-enabled middleware while maintaining failure and replication transparencies for clients provided by Fault Tolerant CORBA object groups.

Figure 1: Overall design rationale

The consistency of server state is not managed by the FT-enabled middleware but is rather managed by middleware independent mechanisms provided/supported by the application. The server state may thus rely on a persistent state service, a replicated database, a Data Distribution Service … etc.

Applications may now perform fault analysis, fault containment and recovery and also provide application-specific fault detectors to complement the FT-middleware enabled fault detector.

To provide failure and replication transparencies to clients, the proposed design makes use of location forwarding techniques to locate a suitable Server replica that is capable of dealing with their requests.

Servers register themselves with a central registry service (Server Group Manager) which maintains records of the endpoints of all Server replicas. The reference advertised to clients contains endpoint information of one or more forwarding components or a location service. When a Client attempts to use a Server's object reference it will therefore be directed to a forwarding service locator instead, which will return the reference for a suitable Server replica that the Client should use. A simplified diagram of this process can be seen below in Figure 2:

Figure 2: Diagram of the basic forwarding mechanism

7.1 General Architecture

CORBA Fault Tolerance architecture tackled most of the architectural elements required for a reliable architecture: entity redundancy, fault detection, and recovery.

By externalizing fault detection and recovery to enable applications use more efficient mechanisms, the proposed architecture focuses on Replication Management and Transparent Failover.

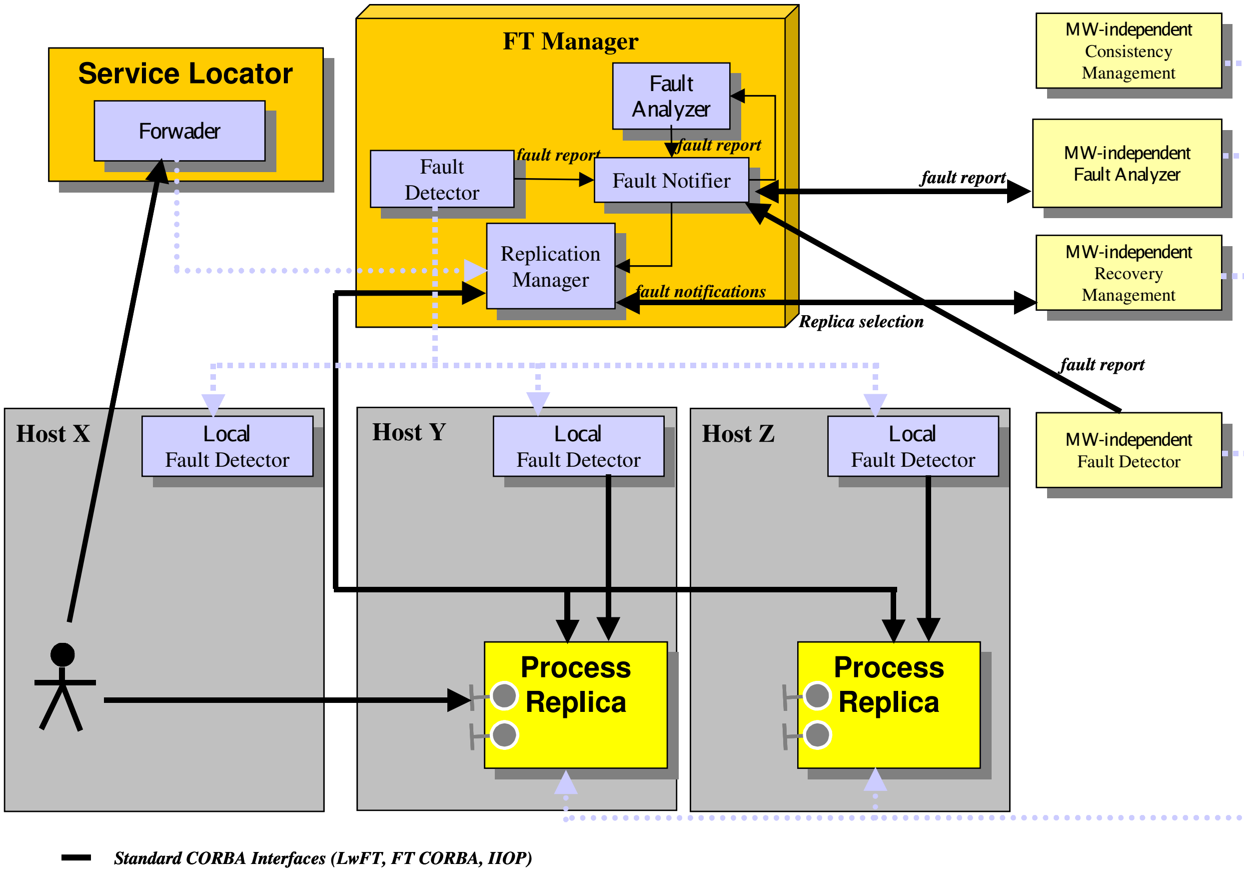

Figure 3: General Architecture

7.1.1 FT-enabled middleware

FT-enabled middleware is responsible for providing the group abstraction needed for our reliable architecture through middleware Replication Management and transparent failover. On primary replica failure, the middleware ReplicationManager may either elect a new one via some predefined strategies, or involve the middleware-independent FT infrastructure or the application for some specific/dedicated election strategy.

The FT middleware may also rely on some middleware fault detectors for monitoring the middleware infrastructure.

A Service Locator architectural component is also introduced to overcome a current FT CORBA limitation when all member replicas have failed (7.1.6).

7.1.2 Middleware-independent FT infrastructure

The FT-enabled middleware relies on external mechanisms for fault detection, fault analysis, and consistency and recovery management.

7.1.3 Server/Process Replication Management Independent from Access to Service

Server process replication done through the **ServerGroupManager** interface does not make any assumption on the way application offer/use services (Client/Server, Data Publish/Subscribe … etc.).

This specification focuses on the CORBA Client/Server interaction model.

7.1.3.1 Client-side view / Object groups

The **ClientFailoverSemanticsPolicy** provides a means for application developers to explicitly control how and if failover occurs. This Policy will be effective in the context of any object reference.

Use of **object groups** when failure and replication transparencies are required based on standard forwarding mechanisms on either standard IORs or Interoperable Object Group references (IOGRs).

7.1.3.2 Server-side view / Process Groups

The ServerGroupManager manages **process groups** in a fault tolerance domain. Since all objects and components within a process group share computing resources, memory … etc. the unit for ensuring consistency is the server process. In passive replication, a server is either primary or backup and as a side effect, all hosted CORBA objects and components are assumed to be primaries or backups.

At start-up, the server registers with the ServerGroupManager of its fault tolerance domain and joins its server group. Middleware-independent mechanisms may be used for consistency management within the server but these should then advise the ServerGroupManager of the completion/readiness of the server.

The ServerGroupManager then, advises the FT-enabled middleware/FT-ORB of the FT status of the server in order to allow/deny request delivery to the server replica depending on whether it is primary or backup.

7.1.4 Fault Tolerance Current

It is necessary to retry request invocation in the presence of failure to provide failure transparency to applications with the risk of performing a request more than once at the server side. Applications wishing to enforce the at most semantics of CORBA invocation can use the ***FTCurrent*** object to retrieve invocation identification and detect repetitive request invocations.

7.1.5 Invocation Timeouts

Invocation timeouts introduced by FTCORBA and CORBA Messaging can still be used with this specification (RelativeRoundtripPolicy, RequestDurationPolicy … etc.).

7.1.6 Location Forwarding

A Service Locator architectural component is responsible for locating/retrieving latest group object reference when all previously known profiles have failed. This submission extends the Interoperable Object Group Reference to support location mechanisms as in DCE-CIOP Location Mechanism ([CORBA] §16.6.1).

7.1.7 Naming Groups

To ease application bootstrapping and service localisation (Location/Forwarding) object groups are named using human readable schemes.

8 Conceptual Model

This submission is based on proven concepts from the CORBA Fault Tolerance specification and basic forwarding techniques used in many ORB vendors' Implementation Repositories (IMR).

CORBA Fault Tolerance defines *fault tolerance domains* containing several hosts (nodes) and many object groups.

A reliable application within a fault tolerance domain will be made of many local applications running on physical nodes. An application offers some ***services*** for use by other applications or users. ***ServiceAccessPoint*** models the client view/access to the service (see AMSM, DMTF/CIM).

QosPolicy is used to capture main real time properties attached to the application or service access point.

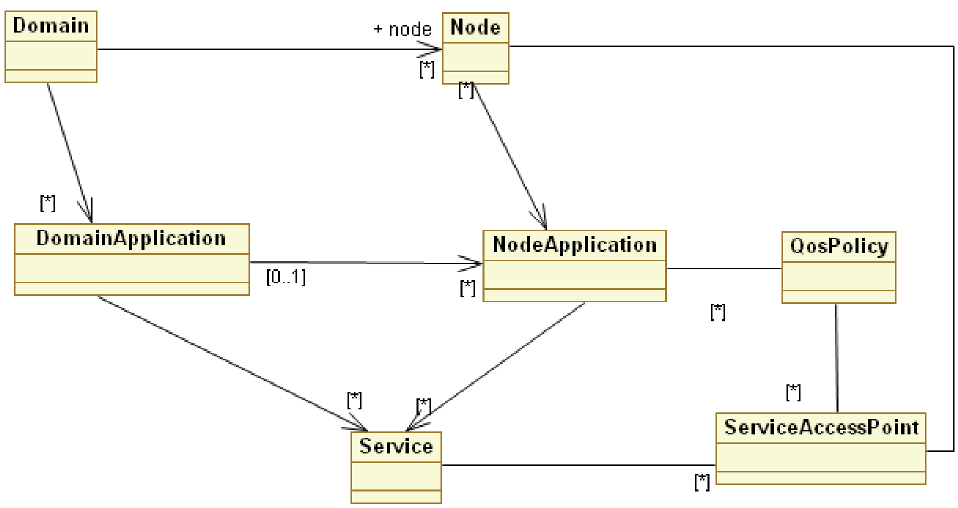

Figure 4: General overview of a Domain Application

The following figure (Figure 5) reuses concepts from DMTF CIM and models a local application in a platform independent way whether services are CORBA based, data centric or another. Only the ***ServiceAccessPoint*** is to be refined for the specific technology.

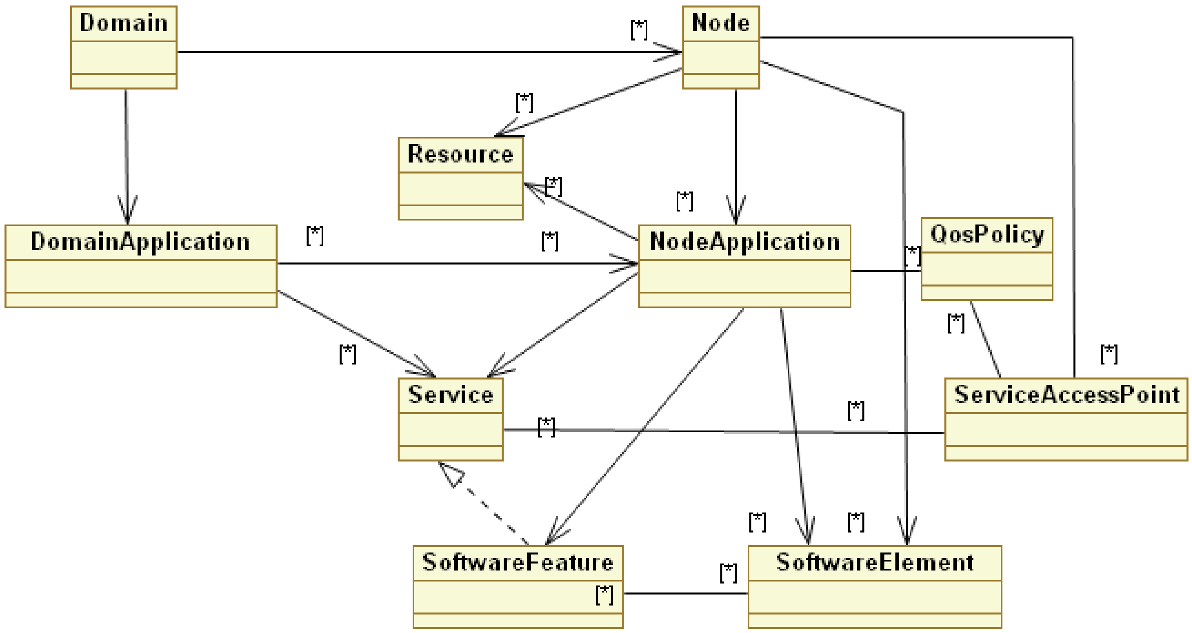

Figure 5: General overview of a Node Application

Class ***SoftwareElement*** (class CIM_SoftwareElement in DMTF/CIM) is a unit of deployment and represents a collection of more files are individually deployed and managed on a particular platform. This may be useful for tackling fault tolerance for CORBA Components.

Class ***SoftwareFeature*** (class CIM_SoftwareFeature in DMTF/CIM) is a unit of component management and represents a collection of software elements that perform a particular function (realizes one or more services).

8.1 Fault Tolerance Domain

CORBA Fault Tolerance [FTCORBA] introduced a fault tolerance domain for improving scalability of large applications where a single Replication Manager manages each fault tolerance domain.

8.2 Redundancy Management

This specification addresses replication at the POSIX process level. So the ***ProcessGroup*** concept models a group of process replicas (***FTProcess***). Each ***FTProcess*** runs on a Node and hosts local ***FTApplication*** local applications (see Figure 6).

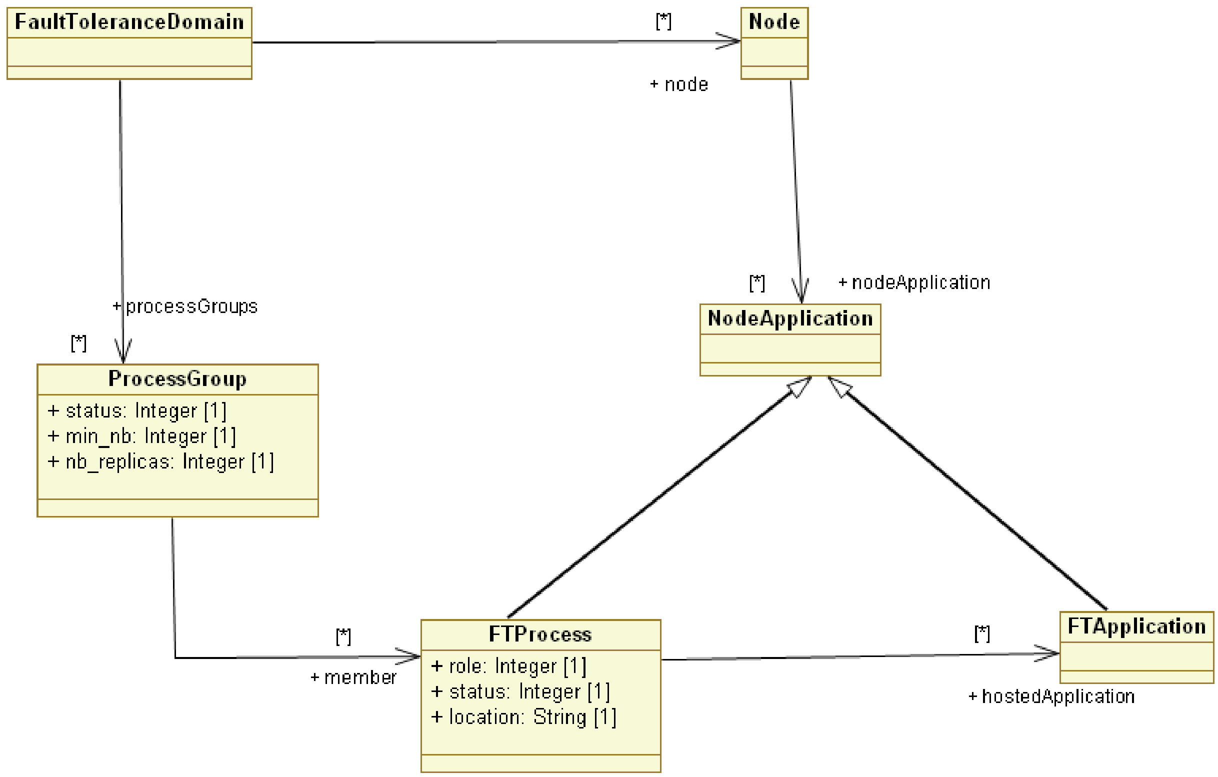

Figure 6: Process Groups and Replicas

A process group is a grouping of node-level POSIX process replicas. At Software Management level, ***CIM_RedundancySet*** collection from DMTF/CIM can be used to model this process group concept.

8.2.1 Process Group

Class **ProcessGroup** has the following two attributes:

- ***min_number_of_replicas***: The minimum number of FTProcess objects desired by the application.
- ***nb_replicas***: the current number of FTProcess objects.

Class specialisation for Passive Replication may provide an additional:

- ***primary_location***: Location of the primary FTProcess.

The ReplicationManager synthesizes the status of a ProcessGroup.

Figure 7: ProcessGroup status lists the status of a ProcessGroup depending on the current number of replicas and the minimum number of replicas.

Example:

- ***Fully Redundant*** : number of replicas >= minimum number of replicas
- ***Degraded Redundancy*** : 1 < number of replicas < minimum number of replicas

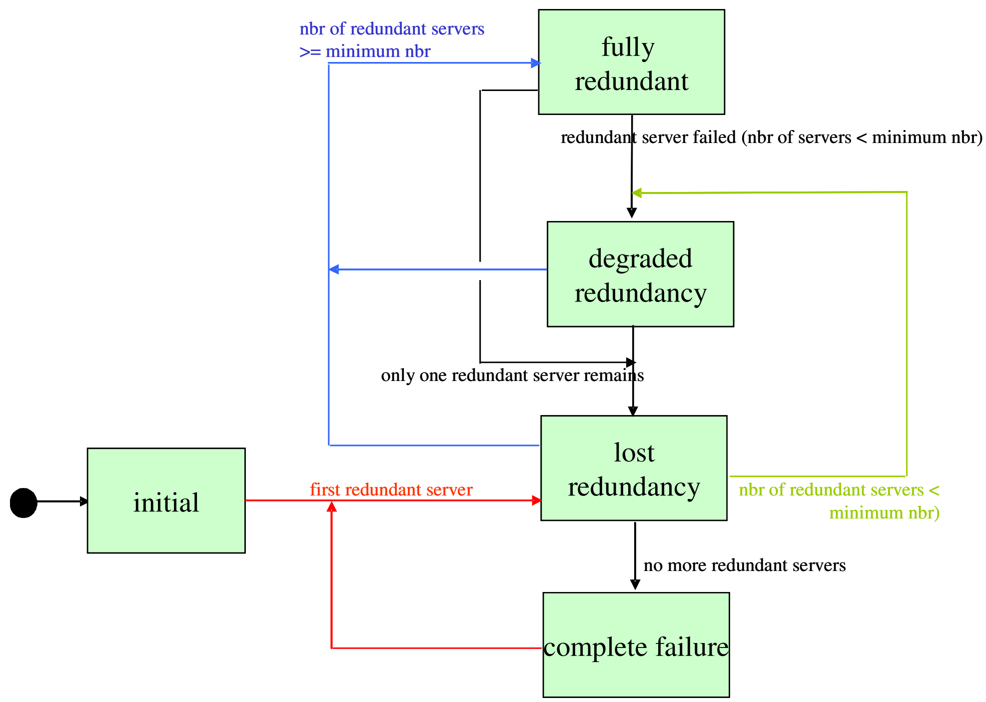

Figure 7: ProcessGroup status

Monitoring of ***ProcessGroup*** objects and possible notifications on the transitions is left to Application Management applications and is thus not in the scope of the FT-enabled middleware.

8.2.2 FTProcess (Process Replica)

The ***FTProcess*** class contains the following attribute:

- ***location***: Location of the replica.

FTProcess Status:

- RECOVERING: The server process is performing its recovery/initialisation to get ready to service request in case of a primary service, or else to become ready to take over should the primary fail in case of a backup server. It is not yet ready to service requests so the FT-enabled middleware will not deliver to it group requests. Non-group requests such as administrative requests will however be delivered to the server in order to allow it to perform any required initializations.
- READY: The server process has now completed its recovery and it is ready to service requests in case of a primary service, or else to become ready to take over should the primary fail in case of a backup server.

FTProcess Replication Role:

- BACKUP: The server process is either backup or recovering to become a backup. Once its status has become ready, the server will be eligible to be become a primary once current primary server fails.
- PRIMARY: The server process is either primary or recovering to become a primary. Once its status has become ready, the server will be serviong group requests.
- ACTIVE: In the case of active replication, the server has an active role and will be allowed to service requests once its status becomes ready.

Figure 8: FTProcess status in PassiveReplication provides the status of an FTProcess class in the case of PassiveReplication.

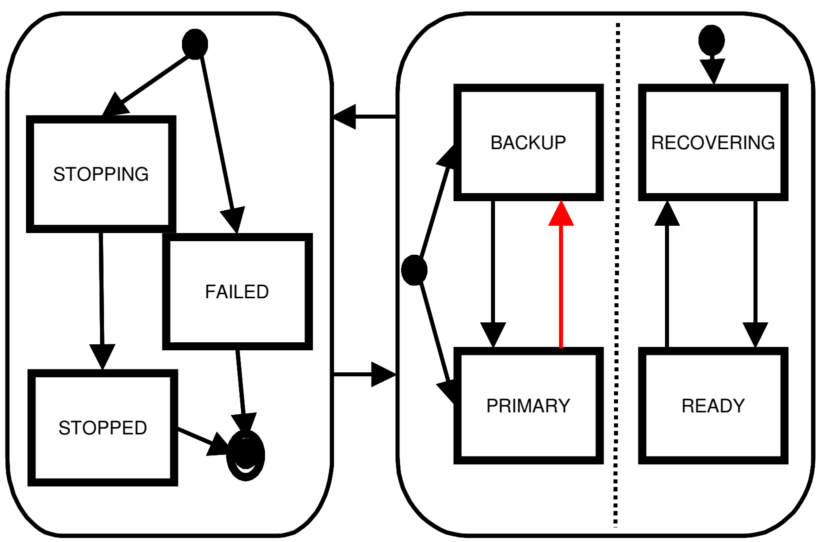

Figure 8: FTProcess status in PassiveReplication

The Process Replica registers with the Replication Manager in order to join its group and is told whether it shall be initializing as primary or backup. At the end of its initialization/insertion, it shall advise the ReplicationManager of its readiness to be allowed to receive requests on the fault tolerant group. See 8.2.4.1 for more details on backup insertion.

8.2.3 Replication Management

The ReplicationManager is an important component of the Fault Tolerance Infrastructure that interacts with other components of the infrastructure. It is responsible for the creation of process groups, object groups for accessing application services and the handling of all group membership requests.

The use of object group identifiers defined in CORBA Fault Tolerance allows for efficient retrieval of object group information. But both process groups and object groups may be named for efficient initialization/bootstrap of middleware independent FT infrastructure and clients and servers.

The FT Server registers with the Replication Manager in order to join its process group and then gets its replication role for performing its recovery (initializing as primary or backup for example). At the end of its recovery, it shall advise the ReplicationManager of its readiness to be allowed to receive requests for the fault tolerant object groups. See 8.2.4.1 for more details on backup insertion.

8.2.4 Passive Replication

In the Warm-passive replication style, only the primary member is executing the methods invoked on the object group by the client objects. The state of the primary is transferred to the backups to be ready for recovery from the primary replica failure.

8.2.4.1 Backup Insertion

Backup insertion refers to the start-up sequence of a replica and its acquisition of the existing primary replica's state in order to be considered as a backup to the primary. Once a backup has been 'inserted', it is eligible for promotion to primary in case of a primary replica failure.

The insertion of a replica may or may not disrupt the functioning of the primary.

The application or any consistency management shall advise the FT-middleware ReplicationManager of the completion of the backup insertion/initialization.

Though not part of this specification, middleware independent consistency management mechanisms are advised to perform the following:

- Failure of a primary replica while backups are inserting should trigger cancellation of their insertion.
- During a backup insertion, the state transfer should not start until the primary has completed its initialization.

8.2.4.2 Recovery

The Backup server is advised by the ReplicationManager of its promotion to Primary in order to start its recovery. Once all state is recovered (responsibility of middleware-independent infrastructure) it shall advise the ReplicationManager of recovery completion in order to generate Server transition to Primary and allow request deliveries.

8.2.5 Active Replication

Basic Redundancy Management is intended for both Passive and Active replication styles. However this specification does not define some group communication protocol for accessing services offered by the FTApplications.

Only a TCP/IP based protocol suitable for passive replication only is addressed by this submission.

8.3 Fault Management

Fault Management provides the following functions:

- 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 or any induced fault reports.
- Fault Containment: preventing the propagation of faults from their origin at one point in a system to a point where it can have an effect on the service to the user.

8.3.1 Fault Containment

Middleware-independent FT infrastructure is responsible for enforcing fail-stop semantics by aborting/killing any server process (FTProcess) that is suspected of being faulty.

Terminating any suspected process will prevent a process considered faulty from sending undesired messages.

8.3.2 Fault Detection & Notification

A FaultNotifier is provided by the FT-enabled middleware for notifying applications of any failures reported to it.

Server process may support a PullMonitorable interface to be used by local fault detectors (middleware independent or FT-enabled middleware) for monitoring the state of the server replica.

Fault Reports conveyed to the Fault Notifier by the Fault Detectors shall conform to CosNotification::StructuredEvent with event type as specified in CORBA Fault Tolerance specification.

8.3.2.1 FaultNotifier

The FaultNotifier [FTCORBA] defines operations for use by the fault detectors and fault analysers to push fault reports for the FaultNotifier. The FaultNotifier propagtes such fault reports to registered entities such the ReplicationManager, and any other fault analyser or application objects.

8.3.3 Fault Analysis/Diagnosis

Once a server process (FTProcess) has failed, all embedded/hosted applications shall be considered faulty.

8.3.4 Connection Management

Support for transport heartbeats may be provided by the FT-enabled middleware to close connections to failed server hosts or with failed communication links.

8.4 QoS Policies

To enforce some real-time properties a set of QoS policies are needed. These apply to different steps/phases of the application.

8.4.1 FTProcess Recovery time

FTProcess recovery (for both Passive and Active replication styles) may only be achieved within this time limit. A failure to comply with this time duration may be reported to applications by the middleware independent FT infrastructure.

It is the responsibility of middleware independent fault analyzer to decide whether this is an application failure or not, in which case it should enforce the crash/stop model by terminating the FTProcess.

8.4.2 Maximum Service Localization Time

Locating the service shall not exceed this time duration. FT-enabled middleware shall abort localization of the service when this time duration is exceeded.

8.4.3 Service Access Properties

Some QoS properties may be needed for accessing the service (such as CORBA priorities in RT CORBA).

8.4.4 Invocation Maximum Blocking Time

A client accessing the service cannot be blocked for more than the maximum blocking time.

8.4.5 Transport/Connection Timeouts

Transport-level timeouts for (heartbeating, keepalive) may be provided by the FT-enabled middleware to close connections to failed server hosts or with failed communication links.

8.5 Transparent Failover

Failover semantics as defined in the CORBA Fault Tolerance specification proved very efficient and workable. This specification relies on currently defined semantics from [FTCORBA].

The Service Locator (cf. 8.5.3) is responsible for supplying the latest working object reference.

8.5.1 At Most Once Semantics

Enforcing at most once semantics by the means of request and reply logging defined in CORBA Fault Tolerance specification is not mandatory for the FT-enabled middleware as applications have better knowledge of operation semantics and can thus handle this in an efficient manner.

Should a unique request identifier be required by the application in order to detect repetitive requests, the FT Context object (cf. 8.5.2) can provide the necessary information.

8.5.2 FT Context

The FT Context object provides access to invocation information for applications requiring this knowledge for consistency management or enforcing at most once semantics.

8.5.3 Service Locator

The Service Locator is responsible for providing access to replicated objects/components when direct connection between client and server processes is broken. Depending on the kind of reference is used by the FT-enabled middleware; the Service Locator may also be used at startup to open initial connections between client and server processes.

Implementers of the Service Locator are free to access the ReplicationManager, and run any implementation-specific protocol as long as they can assure a predictable time for providing a valid reference.

9 OMG CORBA/IDL Platform Specific Model

9.1 Server Group

A **ServerId**, in the context of LWFT identifies a **ServerGroup**. All servers with a common ServerId are replicas of each other and members of the same ServerGroup.

```
module LWFT
{
        // PortableInterceptor::ServerId is a string
        typedef PortableInterceptor::ServerId ServerId;
};
```

Server ID Is defined in the CORBA specification section **8.5.1.1 (08-01-04) Server ID. This definition is reproduced below:**

“A Server ID must uniquely identify a server to an IMR. This specification only requires unique identification using a string of some kind and does not intend to make more specific requirements for the structure of a server ID. The server ID may be specified by an **ORB_init** argument of the form **–ORBServerId.** The value assigned to this property is a string. All templates created in this ORB will return this server ID in the server_id attribute. It is required that all ORBs in the same server share the same server ID. Specific environments may choose to implement -ORBServerId in ways that automatically enforce this requirement.”

Section **15.2.1 Model Components** of the CORBA specification **(08-01-04)** contains the following definition**:**
“*Server*—A server is a computational context in which the implementation of an object exists. Generally, a server corresponds to a process.” **In LWFT a Server will always correspond to a Process.**

9.2 Server Replica Identity

As in the Fault Tolerant CORBA specification, the Location is used to identify a process instance within a ServerGroup and globally. The FT CORBA specification defines a FT::Location as follows:

“The name for a fault containment region, host, device, cluster of hosts, etc., which may be hierarchical.... For each object group and each location, only one member of that object group may exist at that location.”

However, in LWFT Server process based replication following additional constraint applies:

Each LWFT server process must have only one Location and the Location must uniquely identify the server process. For example, a LWFT server process Location might well include both the DNS host name or IP address and the process id. This way the Location will uniquely identify the server process instance within the fault tolerance domain.

```
module LWFT
{
        typedef PortableGroup::Name Location;
};
```

9.2.1 Server Status

```
        typedef long ServerStatusValue;
        const ServerStatusValue SRV_STATUS_UNKNOWN    = 0;
```

9.2.1.1 SRV_STATUS_RECOVERING

```
        const ServerStatusValue SRV_STATUS_RECOVERING = 1;
```

The server process is performing its recovery/initialisation to get ready to service requests in the case that it is a primary service, or else to become ready to take over should the primary fail in case where it is a backup server.

The server is not yet ready to service requests so the FT-enabled middleware will not deliver group requests to it.

Non-group requests such as administrative requests will however be delivered to the server in order to allow it to perform any required initialisations.

9.2.1.2 SRV_STATUS_READY

```
const ServerStatusValue SRV_STATUS_READY      = 2;
```

The server process has now completed its recovery and it is ready to service requests in the case that it is a primary service, or else to become ready to take over should the primary fail in case that it is a backup server.

9.2.2 Server Replication Role

```
typedef long ServerReplicationRoleValue;
const ServerReplicationRoleValue SRV_REPL_UNKNOWN     = 0;
```

9.2.2.1 SRV_REPL_PRIMARY

```
const ServerReplicationRoleValue SRV_REPL_PRIMARY     = 1;
```

The server process is either primary or recovering to become a primary.

Once its status has become ready, the server will be serving group requests.

9.2.2.2 SRV_REPL_BACKUP

```
const ServerReplicationRoleValue SRV_REPL_BACKUP      = 2;
```

The server process is either backup or recovering to become a backup.

Once its status has become ready, the server will be eligible to be become a primary once current primary server fails.

9.2.2.3 SRV_REPL_ACTIVE

```
const ServerReplicationRoleValue SRV_REPL_ACTIVE      = 3;
```

In the case of active replication, the server has an active role and will be allowed to service requests once its status becomes ready.

9.3 Naming Object Groups

The **ObjectGroupName** is a hierarchic **CosNaming::Name**. Name Components within the name must be ordered according to the following schema:

```
<server_id>.SRV/<orb_id>.ORB/<poa_id>.POA[/<sub_poa_id>.POA*]/<object_id>.OBJ
```

where poa_id is the first named POA below RootPOA and sub_poa values are in order of occurrence beneath that POA.

A shorthand notation may be supported when there is no ambiguity:

```
<server_id>/<object_id>
```

9.3.1 Support for Aliases

Applications should be able to provide aliases for object group names to allow for more application-oriented naming schemes. The object group name (alias) is unique within the context of the Fault Tolerance domain.

One way to define this alias when the object group is created via a PortableGroup::GenericFactory interface is to define a new ft property à la **'org.omg.ft.alias'** to be provided in the criteria parameter of **create_object** operation. The following default alias should be made available within the fault tolerance domain with an ObjectGroupManager:

```
<object_group_id>.GID
```

Administrative tools may also be used for naming the object groups via implementation-specific interfaces.

```
module LWFT  {
        const string FT_ALIAS = "org.omg.ft.Alias";
};
```

9.4 ClientFailoverSemanticsPolicy

The **ClientFailoverSemanticsPolicy** provides a means for application developers to explicitly control how and if failover occurs. This Policy will be effective in the context of any object reference.

```
module LWFT {

	enum ClientFailoverSemanticsPolicyValue {
			ORB_DEFAULT,
			BEST_EFFORT,
			AT_LEAST_ONCE,
			AT_MOST_ONCE
	};
	local interface ClientFailoverSemanticsPolicy : CORBA::Policy {
			readonly attribute ClientFailoverSemanticsPolicyValue value;
	};
};
```

The four values of this Policy modify the client ORB behaviour as follows:

- **ORB_DEFAULT** - The ORB behaviour is unmodified from its default.
- **BEST_EFFORT** - This value of the Policy provides the client application with a guaranteed 'best effort' delivery attempt. Normal CORBA invocation semantics do not require that a compliant ORB **must** reinvoke requests on COMPLETED_NO retryable exception conditions or exhaust all available addresses in an IOR. Compliant client ORBs must provide the following behaviour under the influence of this Policy value: Each time a client ORB attempts to invoke a method, it must not abandon the invocation and raise an exception to the client application until it has tried to invoke the server using all of the alternative addresses in the IOR, or has received a “non-failover” condition, or the request duration has expired.
- **AT_LEAST_ONCE** - This value behaves as BEST_EFFORT however retryable exception conditions with status COMPLETED_MAYBE must also be reinvoked.
- **AT_MOST_ONCE** - This value behaves as AT_LEAST_ONCE but the client ORB must transmit an FT_REQUEST service context (***FTRequestServiceContext)*** with the request.

9.5 ObjectGroupMembershipPolicy

The **ObjectGroupMembershipPolicy** POA policy is to be used on the server side to control object group management in a portable way.

Application developers can specify this policy on POA creation to indicate to the FT enabled ORB that Objects created on that POA should automatically be members of replicated Object Groups and that the references returned from POA operations should be ObjectGroup references.

```
module LWFT {

    enum ObjectGroupMembershipPolicyValue
    {
            AUTO_ORB_CTRL,
            AUTO_OBJ_GRP_MNGR,
            USER_CTRL_OBJ_GRP_MNGR
    };

    local interface ObjectGroupMembershipPolicy : CORBA::Policy
    {
            readonly attribute ObjectGroupMembershipPolicyValue value;
    };
};
```

The three ordered values of ObjectGroupMembershipPolicy represent increasing levels of user control over the mechanisms used to provide fault tolerance and corresponding decreasing user application transparency.

ObjectGroupMembershipPolicy values are discussed in the following sections.

9.5.1 AUTO_ORB_CTRL

The objects references created on the POA will be automatically FT replicated across all similar server processes (i.e. that have a matching FT POA hierarchy). Every object reference created on such a POA will be made a member of the ObjectGroup named as per the naming scheme defined in Section 2.4.3 Naming Object Groups. i.e.:

<server_id>.SRV/<orb_id>.ORB/<poa_id>.POA[/<sub_poa_id>.POA*]/<object_id>.OBJ

The behaviour from the users perspective is ' **AUTO_OBJ_GRP_MNGR** (see below) but the mechanism used by the server side ORB to ensure this is left to the ORB implementation.

N.B.: This policy value is only valid on POAs with LifespanPolicy 'PERSISTENT' and IdAssignmentPolicy 'USER_ID'.

All POA operations that create object references will return references that refer to the whole object group rather than to this individual member.

9.5.2 AUTO_OBJ_GRP_MNGR

The objects created on the POA will be automatically FT replicated as above and as per the behaviour below. The POA will transparently use the relevant ObjectGroupManager operations to ensure that this is achieved. At the point that a reference is created on a POA with this Policy value (and before the reference is returned to the user code) the POA should make use of appropriate GOA / ObjectGroupManager operations to:

1. Get the group to which this object should belong
2. Add the member to it.

3. Return the new IOGR to the caller of the POA operation.

N.B.: Only valid on POAs with IdAssignmentPolicy USER_ID. All POA operations that create object references will return references that refer to the whole object group rather than to this individual member.

To retrieve the object group reference, compliant POAs should first use the object group Service**Locator**, and if no ServiceLocator is present, it shall attempt to use the **NameService**.

Compliant ORBs / FT implementations should make the ServiceLocator available to the POA by **ORB::resolve_initial_references ("FTServiceLocator")**.

9.5.3 USER_CTRL_OBJ_GRP_MNGR

The POA created with this policy will be a GOA. This will make the GOA object group management operations available to the application developer. The interaction with the FT infrastructure will be via the ObjectGroupManager as per the already existing FT CORBA spec.

Compliant ORBs / FT implementations should make the ObjectGroupManager available to the Object Adapter by **ORB::resolve_initial_references ("FTObjectGroupManager").**

9.6 ObjectGroupNameResolutionPolicy

The **ObjectGroupNameResolutionPolicy** POA policy is needed in conjunction with AUTO_OBJ_GRP_MNGR value of ObjectGroupMembershipPolicy policy to select a fully qualified naming scheme or a shorthand notation.

```
module LWFT {
    enum ObjectGroupNameResolutionPolicyValue
    {
        USE_FULLY_QUALIFIED_NAME,
        USE_SHORTHAND_NOTATION
    };
    local interface ObjectGroupNameResolutionPolicy : CORBA::Policy
    {
        readonly attribute ObjectGroupNameResolutionPolicyValue value;
    };
};
```

In the absence of a specified Policy value being defined on the POA the LWFT ORB will assume a default of USE_FULLY_QUALIFIED_NAME.

9.7 FT_REQUEST Service Context

The FT_REQUEST service context (FTRequestServiceContext) is defined in CORBA Fault Tolerance to ensure that a request is not executed more than once under fault conditions.

When **AT_MOST_ONCE** is used for the **ClientFailoverSemanticsPolicy**, the FT-enabled middleware shall encode a CDR encapsulation of the FTRequestServiceContext struct in the context_data of the ServiceContext struct of a request message header.

```
module IOP {
    const ServiceId FT_REQUEST = 13;
};
module FT {
    struct FTRequestServiceContext { // context_id = FT_REQUEST;
            string client_id;
            long retention_id;
            TimeBase::TimeT expiration_time;
    };
};
```

The FT_REQUEST service context contains a unique client_id for the client, a retention_id for the request, and an expiration_time for the request. The ***client_id*** and ***retention_id*** serve as a unique identifier for the client's request and allow the server application to recognize that the request is a repetition of a previous request. A server application willing to enforce at most once semantics on its incoming requests may choose to return a previously generated reply in case of repetition of previous requests.

The ***expiration_time*** serves as a garbage collection mechanism. It provides a lower bound on the time until which the server application must honor the request and, therefore, retain the request and corresponding reply (if any) in its log.

9.8 FTRequestDurationPolicy

The Request Duration Policy is defined in CORBA Fault Tolerance. It determines how long a request, and the corresponding reply, should be retained by a server to handle reinvocation of the request under fault conditions.

```
module FT {
    const CORBA::PolicyType REQUEST_DURATION_POLICY = 47;
    interface RequestDurationPolicy : CORBA::Policy {
            readonly attribute TimeBase::TimeT request_duration_policy_value;
    };
};
```

The Request Duration Policy, applied at the client, defines the time interval over which a client's request to a server remains valid and must be retained by the server application to detect repeated requests.

The ***request_duration_policy_value*** is added to the client ORB's current clock value to obtain the ***expiration_time*** that is included in the FT_REQUEST service context for the request.

9.9 Transport Heartbeats

'23.2.9 Transport Heartbeats' of the CORBA specification defines TAG_FT_HEARTBEAT_ENABLED component of the TAG_INTERNET_IOP profile, and two policies: Heartbeat and HeartbeatEnabled to solve the problem of host and/or link failures in IIOP (TCP/IP).

This specification reuses those mechanisms. Fully compliant Server ORBs should be able to respond to the FT_HB pseudo operation as defined in that section. In the case of Server ORBs that are not compliant, users may choose to implement this operation at the application level. Alternatively, components of the LWFT middleware can identify Servers that are not fully compliant by a BAD_OPERATION.

9.9.1 TAG_FT_HEARTBEAT_ENABLED Component

"The TAG_FT_HEARTBEAT_ENABLED component in a TAG_INTERNET_IOP profile indicates that the addressed endpoint supports heartbeating."

```
module IOP {
    const ComponentId TAG_FT_HEARTBEAT_ENABLED = 29;
};
module FT {
    struct TagFTHeartbeatEnabledTaggedComponent {
            // tag =TAG_FT_HEARTBEAT_ENABLED
            boolean heartbeat_enabled;
    };
};
```

9.9.2 Heartbeat Policy

"The Heartbeat Policy, applied at the client, allows the client to request heartbeating of its connections to servers, using the heartbeat_interval and heartbeat_timeout."

```
module FT {
    const CORBA::PolicyType HEARTBEAT_POLICY = 48;
    struct HeartbeatPolicyValue {
            boolean heartbeat;
            TimeBase::TimeT heartbeat_interval;
            TimeBase::TimeT heartbeat_timeout;
    };
    interface HeartbeatPolicy : CORBA::Policy {
            readonly attribute HeartbeatPolicyValue heartbeat_policy_value;
    };
};
```

"When the Heartbeat Policy is applied at a client ORB, the ORB is responsible for taking the following steps. While a

connection exists to a remote server, the ORB sends a request message over the connection at least as often as was requested by the heartbeat_interval of the Heartbeat Policy of any client connected to a server over that connection. The request message is equivalent to an invocation of the method:

```
void FT_HB ();
```

on any one of the server objects accessed by the connection…"

9.9.3 Heartbeat Enabled Policy

"Because heartbeating can generate significant network traffic, and can use significant server resources, the heartbeating capability is explicitly enabled or disabled using the Heartbeat Enabled Policy."

```
module FT {
        const CORBA::PolicyType HEARTBEAT_ENABLED_POLICY = 49;
        interface HeartbeatEnabledPolicy : Policy {
                readonly attribute boolean heartbeat_enabled_policy_value;
        };
};
```

"The Heartbeat Enabled Policy allows the heartbeating of a server endpoint. If the Heartbeat Enabled Policy is enabled for a server endpoint, the TAG_INTERNET_IOP profile for that endpoint contains the TAG_FT_HEARTBEAT_ENABLED component to indicate to the client that the server endpoint is heartbeat_enabled."

9.10 Interoperable Object Group References

The object group abstraction is very powerful and allows for the provision of both failure and replication transparencies for clients. This specification extends the existing FT CORBA definition of Interoperable Object Groups with the support for Location agents.

Use of FT_REQUEST service context allows applications that require enforcing at most once semantics to access (via FT Current interface) request information.

Use of FT_GROUP_VERSION service context allows a server to determine whether a client is using an obsolete object group reference and, if so, to respond with the most recent object group reference to update the client's object reference.

This specification introduces a new FT_GROUP_ID service context to allow for more efficient lookup of latest IOGR in case of requests with older versions coming to the process as the object group identifier may not necessarily be obtained from the object key. This avoids comparison of IORs as it is more likely to have more than one object group with a local member.

9.10.1 Support for Location Agents.

Current FT CORBA states the following in §23.2.4 Modes of Profile Addressing:

"The interoperable object group references contain profiles that address server object groups. This section illustrates the use of these profiles according to one of two modes:

- *Profiles that address object group members.*
- *Profiles that address gateways (technically generic in-line bridges of the type described in the Building Inter-ORB Bridges chapter of the CORBA specification)."*

To allow for the support of Location services as in FT CORBA §16.6.1 Location Mechanism Overview, the following shall be added:

- Profiles for agents that may not be able to accept direct requests for any objects, but acts instead as a location service. Any request messages sent to the agent would result in either exceptions or replies with LOCATION_FORWARD status, providing new addresses to which requests may be sent. Such agents would also respond to locate request messages with appropriate locate response messages.
- Profiles for agents that may directly respond to some requests (for certain objects) and provide forwarding locations for other objects.

9.10.2 FT_GROUP_ID Service Context

The FTGroupIdServiceContext struct contains the version of the identifier of the server object group, to assist the server in getting the object group identifier when not encoded within the object key. When encoded in a request or reply message header, the context_data component of the ServiceContext struct shall contain a CDR encapsulation of the FTGroupIdServiceContext struct, which is defined below.

```
module IOP {
    const ServiceId FT_GROUP_ID = 17;
};
module FT {
    struct FTGroupIdServiceContext { //context_id = FT_GROUP_ID;
            ObjectGroupId object_group_id;
    };
};
```

If the server determines from the FT_GROUP_VERSION service context that the client is using an obsolete object group reference, the server may use the object_group_id value to get the most recent object group reference for the server group and then return it in a LOCATION_FORWARD_PERM response.

```
const ServiceId FT_GROUP_ID = 17;
```

A constant that designates the FT_GROUP_ID service context.

```
struct FTGroupIdServiceContext { //context_id = FT_GROUP_ID;
    ObjectGroupId object_group_id;
};
```

A structure that contains the same object_group_id that is in the TAG_FT_GROUP component of each of the

TAG_INTERNET_IOP profiles of the object group reference for the server object group, to be used, if necessary, by the server ORB to obtain the most recent object group reference of the server object group.

A server ORB that is capable of obtaining the the most recent object group reference of the server object group without the help of the FT_GROUP_ID service context should ignore this context. On the other hand, should the FT_GROUP_ID service context be required by the server ORB but not present in the client's request, the server ORB shall return an INV_OBJREF exception to the client.

9.10.3 Persistent vs. Transient IOR

Depending on the technique used by the FT-enabled middleware for providing middleware transparent failover (between the ReplicationManager and its Gateways), server-side FT-enabled middleware may choose to create transient references only to avoid backup replicas (previously started as primary then restarted) receiving client requests in case that only a primary object reference is used within the object group reference.

Object Group membership operations initiated by the Group Object Adapter (§9.31) will rely on the LifeSpan policy used at the object adapter's creation.

9.10.4 Gateway

CORBA Fault Tolerance specification defines a Gateway for non-FT aware ORBs. This gateway may be extended to provide Service Location capabilities as defined in 8.5.3.

The gateway may rely on LOCATION FORWARDING techniques such as those used in IMRs.

Within a Fault Tolerance domain (managed by a single Replication Manager), the Gateway shall not make any assumption about the client's ORB's ability to assure portability and keep any constraint that may be imposed by the FT-enabled middleware within the 'Server-scope'.

9.11 GenericFactory

FT-enabled middleware relying on **PortableGroup::GenericFactory** interface for the creation of object groups may support naming (also named aliasing in this specification) of object groups in order to ease bootstrapping to applications.

```
module PortableGroup
{
    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);
    }; // end GenericFactory
};
```

Add a new criteria for use by the **GenericFactory::create_object**() operation to hold the Alias property.

9.11.1 Alias property

FT_ALIAS property name = ***'org.omg.ft.Alias'***. When no Alias property is provided at the object group creation a default name will be supplied by the **GenericFactory** : ***“<<object group id>>.gid***”.

9.12 ServerCallback interface

```
interface ServerCallback {};
```

The **ServerCallback** interface is the base for specific callback specialisations. Servers can pass callback references to the FT Middleware to enable the middleware, and actors external to the middleware which are observing the ServerGroup, to interact with the individual ServerGroup process members.

9.13 GroupUpdateObserver interface

GroupUpdateObserver is a specialisation of **ServerCallback** interface for use when it is necessary to be updated whenever the list of process group members changes. Each process group entry contains the location of the member, its callback object and the list of process-specific properties passed to the **ServerGroupAdmin** object during the process registration.

```
struct ServerGroupMember
{
      Location          server_location;
      ServerCallback    server_callback;
      Properties        props;
};
typedef sequence<ServerGroupMember> ServerGroupMembers;
interface GroupUpdateObserver : ServerCallback
{
      void on_update_group(in ServerGroupMembers members);
};
```

9.13.1 on_update_group operation

```
void on_update_group(in ServerGroupMembers members);
```

The ServerGroupManager (or other part of the FT infrastructure) can notify the process of a change of the list of process group members.

9.14 ServerGroupNotFound exception

```
exception ServerGroupNotFound {};
```

The server group with the given identifier is not found by the ServerGroupManager. This exception may be raised if the ServerGroupManager can only register known ServerGroups.

9.15 ServerGroupNotFound exception

```
exception ServerNotFound {};
```

No server process is known by the ServerGroupManager exists for the given location value.

9.16 WrongStatus exception

```
exception WrongStatus {};
```

Operation incompatible with current process status.

9.17 AlreadyRegistered exception

```
exception AlreadyRegistered {};
```

The Server Location value has already been registered..

9.18 UnsupportedCallback exception

```
exception UnsupportedCallback {};
```

The ServerGroupManager does not support provided ServerCallback type.

9.19 Server Group Manager

Compliant FT implementations should register an instance to be accessible to the server ORB via **ORB::resolve_initial_references ("FTServerGroupManager")**.

```
interface ServerGroupManager
{
        void register_server(inout Location   the_location,
                             in ServerId         the_server_id,
                             in Properties       props,
                             in ServerCallback   callback)
                raises (ServerGroupNotFound, AlreadyRegistered,
                             UnsupportedCallback);

        void unregister_server(in Location location)
                raises (ServerNotFound);

        void server_ready(in Location the_location)
                raises (ServerNotFound,
                             WrongStatus);
};
```

9.19.1 register_server operation

```
void register_server(inout Location       the_location,
                     in ServerId                    the_server_id,
                     in Properties                  props,
                     in ServerCallback              callback)
        raises (ServerGroupNotFound, AlreadyRegistered, UnsupportedCallback);
```

The ***register_server*** operation is used to notify the LWFT infrastructure about the existence of a Server and the ServerGroup that it belongs to.

The ***the_location*** argument denotes the location of the server replica. It is expected that the location will contain host and process or process identification. If (and only if) the process provides an empty location then the infrastructure could provide a value.

The ***the_server_id*** argument identifies the Server (Process) group. It is expected to contain "***-ORBServerId"*** value of **ORB_init**.

The ***props*** parameter allows for infrastructure-specific properties that may be required for correct functioning of the application. Some implementations may require, for example, process endpoints as provided by by ***–ORBListenEndpoints***.

The ***callback*** argument allows the LWFT infrastructure to optionally provide additional control or monitoring over the process. The infrastructure can narrow it to specific subcategories.

The ServerGroupManager will raise **ServerGroupNotFound** exception when ***the_server_id*** does not denote an existing process group. When ***location*** has already been registered, **AlreadyRegistered** exception will be raised.

9.19.2 unregister_server operation

```
void unregister_server(in Location the_location)
      raises (ServerNotFound);
```

The ***unregister_server*** operation is used to notify the LWFT of deliberate deregistration of a server from its server group. This is may occur during graceful stop of the application.

The ***location*** argument identifies the location of the server that is to be unregistered.

ServerNotFound exception is raised when *location* does not denote an already registered server.

9.19.3 server_ready operation

```
void server_ready(in Location the_location)
      raises (ServerNotFound,
                WrongStatus);
```

The ***server_ready*** operation is used to notify the LWFT infrastructure that server recovery has completed and that the application is ready to service incoming requests.

The ***location*** argument identifies the location of the server ready to service FT requests.

ServerNotFound exception is raised when *location* does not denote an already registered server.

WrongStatus exception is raised when the server is in a status that does not expect calls to the server ready operation (this may occur when the server calls server_ready multiple times).

9.20 ServerGroupObserver

ServerGroupObserver interface is an interface actors external to the middleware should implement to be made aware of server registration (i.e. creation); unregistration (i.e. destruction / failure); and server readiness and un-readiness

```
interface ServerGroupObserver
{
    void on_register_server (in Location       the_location,
                             in ServerId       the_server_id,
                             in Properties     props,
                             in ServerCallback callback);
    void on_unregister_server (in Location the_location);
    void on_server_ready (in Location the_location);
};
```

9.20.1 on_register_server operation

```
void on_register_server (in Location       the_location,
                         in ServerId       the_server_id,
                         in Properties     props,
                         in ServerCallback callback);
```

This operation is called whenever a server at location '**the_location**' is registering at the ServerGroupManager joining the server group 'the_server_id'.

9.20.2 on_unregister_server operation

```
void on_unregister_server (in Location the_location);
```

This operation is called whenever a server at location '**the_location**' is unregistering from its server group at the ServerGroupManager.

9.20.3 on_server_ready operation

```
void on_server_ready (in Location the_location);
```

This operation is called whenever a server at location '**the_location**' is calling server_ready operation of the ServerGroupManager.

9.21 Server Group Manager Ext

This extension to ServerGroupManager allows actors external to the middleware to be made aware of server registration (i.e. creation); unregistration (i.e. destruction / failure); and server readiness and un-readiness. By registering a reference to an implementation of ServerGroupObserver

The FT middleware ServerGroupManagerExt will notify all registered observer instances of the ServerGroupObserver interface, in order of registration, when one of its own ServerGroupManager operations have been invoked, by calling the same method on all listeners and passing on the values that it received itself.

```
module LWFT
{
    interface ServerGroupManagerExt : ServerGroupManager
    {
        // Empty sequence means all
        void register_observer (in ServerIdSeq server_groups,
                                in ServerGroupObserver observer);

        void unregister_observer (in ServerIdSeq server_groups);
    };
};
```

9.21.1 Implicit operation register_server

Upon an invocation register_server from an client the ServerGroupManagerExt should call on_register_server on all registered observers. This should be done prior to returning a response to register_server to the client.

If an exception is encountered on any listener register_server invocation then the ServerGroupManagerExt should call on_unregister_server on all observers that the call succeeded on (ignoring any exceptions generated from these on_unregister_server calls) before propagating the exception back to the client.

9.21.2 Implicit operation unregister_server

Upon an invocation of unregister_server from a client the ServerGroupManagerExt should call on_unregister_server on all registered ServerGroupObserver observers. This should be done prior to returning a response to register_server to the client.

If an exception is encountered on any observer on_unregister_server invocation then the ServerGroupManagerExt should call on_unregister_server on all remaining observers (ignoring any further exceptions generated from those on_unregister_server calls) before propagating the original exception back to the client.

9.21.3 Implicit operation server_ready

Upon an invocation of server_ready the ServerGroupManagerExt should call the corresponding call to all registered observers.

9.22 RecoverableServer Interface

Observer interface for being notified whenever recovery is in progress in the case of primary/backup passive replication, i.e., a backup is being promoted to become a primary.

```
module LWFT
{
    interface RecoverableServer : ServerCallback
    {
        void start_recovery (in Properties props);
    };
};
```

9.22.1 start_recovery operation

This callback operation instructs the backup replica that recovery is starting. The infrastructure can call this method to notify the server that it should perform the actions needed to make it ready to accept requests as a primary, for example.

When the server is ready to accept requests it should signal the infrastructure by calling**ServerManager::server_ready**().

9.23 ServerRecoveryManager interface

```
module LWFT
{
    local interface ServerRecoveryManager : ServerManager
    {
        boolean register_recovery_observer (in RecoveryObserver rec);
        boolean unregister_recovery_observer (in RecoveryObserver  rec);

        readonly attribute ServerStatusValue         replica_status;
        readonly attribute ServerReplicationRoleValue replica_role;
    };
};
```

9.24 ServerManager

ServerManager is a local interface that lets the application manage its ServerGroup registration, de-registration and notify the FT infrastructure of its readiness to receive requests.

Compliant ORBs / FT implementations should make this available to the application developer by **ORB::resolve_initial_references ("FTServerManager").**

```
local interface ServerManager
{
    attribute Location the_location;
    void register_server ();
    void server_ready ();
    void unregister_server ();
};
```

9.24.1 the_location attribute

```
attribute Location the_location;
```

This value will default to -ORBFTLocation if specified.Attempting to set a an alternate-ORBFTLocation value on more than one ORB initialisation in process is a user error and the behaviour is undefined. The value of the the_location attribut may not be changed after the opertion register_server has ben called.

9.24.2 register_server operation

```
void register_server ();
```

This call prompts the ServerManager to register the server with the LWFT ServerGroupManager

9.24.3 server_ready operation

```
void server_ready ();
```

This call will notify the LWFT ServerGroupManager that this server is now fully initialised and ready to receive requests.

9.24.4 unregister_server operation

```
void unregister_server ();
```

The application may call this to prompt the call to unregister_server on the ServerGroupManager.

9.25 FT Current

This local interface provides access to request context information (*client_id*, *retention_id* and *expiration_time*). Applications can retrieve its object reference by calling ***ORB::resolve_initial_references("FTCurrent").***

Server implementations can use this information to identify a re-invocation of a request when at most once semantics need to be ensured.

```
module FT {
        // resolve_initial_references("FTCurrent");
        local interface Current : CORBA::Current {

                exception NoContext { };

                string get_client_id()
                        raises(NoContext);
                long get_retention_id()
                        raises(NoContext);
                TimeBase::TimeT get_expiration_time()
                        raises(NoContext);
        };
};
```

9.25.1 NoContext exception

```
exception NoContext { };
```

Exception that indicates a Current operation was invoked outside of an FT invocation.

9.25.2 get_client_id operation

```
string get_client_id() raises(NoContext);
```

Returns the client ID extracted from **FT_REQUEST** service context. **NoContext** exception will be raised if the operation is called outside of an FT operation.

9.25.3 get_retention_id operation

```
long get_retention_id() raises(NoContext);
```

Returns the retention ID extracted from **FT_REQUEST** service context. **NoContext** exception will be raised if the operation is called outside of an FT operation.

9.25.4 get_expiration_time operation

```
TimeBase::TimeT get_expiration_time() raises(NoContext);
```

Returns the expiration time extracted from **FT_REQUEST** service context. **NoContext** exception will be raised if the operation is called outside of an FT operation.

9.26 FaultNotifier

This submission uses the CORBA Fault Tolerance definition of FaultNotifier (see [FTCORBA]).

Failure reports are conveyed to the Fault Notifier by the middleware independent Fault Detectors and by the Fault Notifier to the entities that have been registered for such notifications. The Fault Detectors and Fault Notifier use a well-defined event type to convey a given failure event.

This specification defines an event type that is understood by the FT-enabled fault tolerance middleware. Vendors or the OMG may extend this to convey other failure or fault event types; but only the following failure event type will guarantee portability over FT-enabled infrastructures.

Middleware-independent mechanisms are responsible for enforcing the fail-stop semantics for the failed location.

9.27 ServerCrashFault

The ServerCrashFault event type reports that a given process at a given location has crashed. The definition for the event type is as follows:

```
CosNotification::StructuredEvent fault_event;
fault_event.header.fixed_header.event_type.domain_name = "FT_CORBA";
fault_event.header.fixed_header.event_type.type_name = "ServerCrashFault";
fault_event.filterable_data_length(2);
fault_event.filterable_data[0].name = "FTDomainId";
fault_event.filterable_data[0].value = /* Value of FTDomainId bundled into any */;
fault_event.filterable_data[1].name = “Location”;
fault_event.filterable_data[1].value = /* Value of Location bundled into any */;
```

The **filterable_data** part of the event body contains the identity of the crashed process as a pair of name-values: the fault tolerance domain identifier and the process's location identifier.

9.28 RecoveryObserver

Observer interface for being notified whenever recovery is in progress in the case of primary/backup passive replication,. i.e., a backup is being promoted to become a primary.

```
module LWFT {
    local interface RecoveryObserver
    {
        /**
        * Instructs the backup replica that recovery is starting.
        */
        void on_recovery();
    };

};
```

9.29 Fault Detection

9.29.1 PullMonitorable interface

CORBA Fault Tolerance [FTCORBA] defines a ***PullMonitorable*** interface for use by pull-based Fault Detectors. FT-enabled middleware Fault Detectors may ping periodically process callback objects supporting the **PullMonitorable** interface by invoking the **is_alive()** operation.

```
module FT {
    interface PullMonitorable
        boolean is_alive();
    };
};
```

9.29.1.1 is_alive operation

```
boolean is_alive();
```

"This operation informs the pull-based Fault Detector whether the object is able to accept requests and produce replies. The monitored object may return true directly to indicate its liveness, or it may perform an application-specific "health" check (for example, assertion check) within the operation and return false if the test shows that the object is in an inconsistent state.

Return Value

Returns true if the object is alive and ready to take further requests, and false otherwise."

9.29.2 PullMonitorableServer interface

A ServerCallback specialisation for monitoring the health of the FT process by the FT-enabled middleware .

```
module LWFT
{
    interface PullMonitorableServer : FT::PullMonitorable,
                                      ServerCallback
    {
    };
};
```

9.30 ObjectGroupManager

The Portable Object Adapter may use the **PortableGroup::ObjectGroupManager** for group membership operations as specified in the Fault Tolerant CORBA when created with **AUTO_OBJ_GRP_MNGR** or **USER_CTRL_OBJ_GRP_MNGR** values for ObjectGroupMembershipPolicy.

When **AUTO_OBJ_GRP_MNGR** is used for POA creation, the POA should first resolve the object group name using the **Locator** interface or from the **NameService** when no Locator is available.

```
module PortableGroup {
    // Specification of ObjectGroupManager Interface
    interface ObjectGroupManager {
            ObjectGroup create_member
                            (in ObjectGroup object_group,
                            in Location the_location,
                            in _TypeId type_id,
                            in Criteria the_criteria)
                    raises (ObjectGroupNotFound,
                            MemberAlreadyPresent,
                            NoFactory,
                            ObjectNotCreated,
                            InvalidCriteria,
                            CannotMeetCriteria);
            ObjectGroup add_member
                            (in ObjectGroup object_group,
                            in Location the_location,
                            in Object member)
                    raises (ObjectGroupNotFound,
                            MemberAlreadyPresent,
                            ObjectNotAdded);
            ObjectGroup remove_member
                            (in ObjectGroup object_group,
                            in Location the_location)
                    raises (ObjectGroupNotFound, MemberNotFound);
            Locations locations_of_members
                    (in ObjectGroup object_group) raises(ObjectGroupNotFound);
            ObjectGroupId get_object_group_id
                    (in ObjectGroup object_group) raises(ObjectGroupNotFound);
            ObjectGroup get_object_group_ref
                    (in ObjectGroup object_group) raises(ObjectGroupNotFound);
            Object get_member_ref
                            (in ObjectGroup object_group,
                            in Location loc)
                    raises(ObjectGroupNotFound, MemberNotFound);
    }; // end ObjectGroupManager
}; // end of PortableGroup
```

9.31 Group Object Adapter

The Group Object Adapter (GOA) specified in the MIOP specification may also be used for encapsulating group membership for CORBA objects/components. Use of **AUTO_OBJ_GRP_MNGR** POA policy at POA creation shall create a Group Object Adapter. The GOA may perform automatic calls to add_member() and remove_member() on the ObjectGroupManager.

The ObjectGroupManager may be obtained using ***ORB.resolve_initial_references("FTObjectGroupManager");***

```
module PortableServer {

    exception NotAGroupObject {};
    typedef sequence <PortableServer::ObjectId> IDs;

    interface GOA : ::PortableServer::POA {
        PortableServer::ObjectId
        create_id_for_reference(in Object the_ref)
            raises (NotAGroupObject);
        IDs
        reference_to_ids (in Object the_ref)
            raises (NotAGroupObject);
        void
        associate_reference_with_id
            (in Object ref, in PortableServer::ObjectId oid)
            raises(NotAGroupObject);
        void
        disassociate_reference_with_id
             (in Object ref, in PortableServer::ObjectId oid)
            raises(NotAGroupObject);
    }; // end interface GOA
};
```

9.31.1 create_id_for_reference operation

```
PortableServer::ObjectId
create_id_for_reference(in Object the_ref)
    raises (NotAGroupObject);
```

The operation create_id_for_reference() takes as an argument a widened Group IOR and generates a unique ObjectId for that reference. This returned ObjectId is later associated with a servant via the standard API in the POA; that is, activate_object_with_id().

9.31.2 reference_to_ids operation

```
typedef sequence <PortableServer::ObjectId> IDs;
IDs reference_to_ids (in Object the_ref)
    raises (NotAGroupObject);
```

The operation reference_to_ids() takes as an argument a widened Group IOR and returns a sequence of ObjectIds that are currently associated with the Group IOR. To be resilient to hardware failures only one ObjectId is allowed.

9.31.3 associate_reference_with_id operation

```
void
associate_reference_with_id
        (in Object ref, in PortableServer::ObjectId oid)
    raises(NotAGroupObject);
```

The operation takes a previously generated ObjectId and associates it with a group reference. Servants activated using this ObjectId will be candidates for receiving FT group requests via the group information provided in the IOR. To be resilient to hardware failures only one ObjectId is allowed. The operation silently ignores repeat/duplicate associations of a POA/ObjectId pair with the provided object reference.

This operation performs an automatic call to add_member() operation on the ObjectGroupManager. Compliant ORBs / FT implementations should make the ObjectGroupManager available to the Object Adapter by **ORB::resolve_initial_references ("FTObjectGroupManager").**

9.31.4 disassociate_reference_with_id operation

```
void
disassociate_reference_with_id
        (in Object ref, in PortableServer::ObjectId oid)
    raises(NotAGroupObject);
```

The operation takes a previously generated ObjectId and removes the association it had with a group reference. Servants activated using this ObjectId will no longer receive FT group requests via the group information provided in the IOR. The operation silently ignores disassociations that no longer or never existed.

9.32 FT CCM Component

Use of object group for components provider ports (facets, supported interface and push consumer ports). The Container shall handle group membership operations.

9.32.1 Navigation/Introspection

Used to retrieve and connect ports. To be able to connect a group of components, those navigation and events interfaces have to return group references (IOGR) when required, but also individual port references (IOR).

If a group reference to a navigation or event interface is used, a group reference to the port should be returned.

If an individual/non-group reference to a navigation or event interface is used, an individual reference to the port should be returned.

Selecting whether a group reference or an individual reference is used, CORBA Component Containers may rely on the **FTCurrent** object.

9.32.2 Component Activation

Components that are required to be notified upon promotion of their hosting server process from backup to primary shall support a **Recoverable** interface.

9.33 At most once semantics

Cf. 8.5.1

FT-aware CCM containers shall provide access to the **FTCurrent** object through standard **CCM2Context:: resolve_service_reference(“FTCurrent**”) operation.

9.34 TransportProperties

To allow for a predictable recovery time, applications may require some control on transport protocol parameters. Real Time CORBA [RTCORBA] specifies a TCPProtocolProperties interface to allow the configuration of TCP protocol specific configurable parameters.

```
local interface TCPProtocolProperties : ProtocolProperties {
        attribute long send_buffer_size;
        attribute long recv_buffer_size;
        attribute boolean keep_alive;
        attribute boolean dont_route;
        attribute boolean no_delay;
};
```

Non-RT CORBA ORBs however shall provide a means to enable use of keepalive property on TCP transport. The **–ORBKeepAlive** option to ORB_Init() shall activate the keepalive on open TCP connections. ORB implementers are free to enable keepalive on TCP connections by default.

9.35 ServiceLocator interface

The Service**Locator** interface enables an **ObjectGroup** to be obtained from the **ObjectGroupName** that identifies the ObjectGroup.

```
module LWFT
{
    interface ServiceLocator
    {
            PortableGroup::ObjectGroup locate (in ObjectGroupName group)
                    raises (PortableGroup::ObjectGroupNotFound);

            attribute ServiceLocator fallback;
    };
};
```

The ServiceLocator interface may also be used by the POA (in the case when local interface for use by the POA (in the case of AUTO_OBJ_GRP_MNGR ObjectGroupMembershipPolicy) to locate the ObjectGroup reference needed for group membeship via the PortableGroup::ObjectGroupManager interface.

Compliant ORBs / FT implementations should make this available to the Object Adapter by **ORB::resolve_initial_references("FTServiceLocator").**

9.35.1.1 locate operation

```
PortableGroup::ObjectGroup locate (in ObjectGroupName group)
        raises (PortableGroup::ObjectGroupNotFound);
```

This operation will return the object group reference of the ObjectGroup named by the ObjectGroupName or throw a ObjectGroupNotFound exception if the name is not recognised.

9.35.1.2 fallback attribute

```
attribute Locator fallback;
```

If this attribute is not nil then the Locator will try the locate() on this Locator instead of raising ObjectGroupNotFound.

9.36 ForwardingServiceLocator

The ForwardingServiceLocator is an interface that extends the behaviour of Service Locator with two pseudo operations as follows:

```
module LWFT
{
    interface ForwardingServiceLocator : Locator
    {
            // Pseudo operations
            // FT::ObjectGroup FT_Locate ();
            // Requests made to FT_Locate should invoke any
            // registered RequestDecoders and will then call
            // ::locate returning the located object to
            // the caller.

            // <any other request>
            // Will invoke any registered RequestDecoders and will
            // then call ::locate returning a LOCATION_FORWARD
            // response to the located object to the caller.
    };
};
```

The proposed mechanism makes use of location forwarding techniques to locate and return the latest ObjectGroup reference for a Client.

Object group references created by the FT-enabled middleware should at minimum contain a profile with a ForwardingServiceLocator endpoint information.

Fully compatible ORBs may contact the ForwardingServiceLocator component by and send a request message using the reserved operation name 'FT_Locate'. FT_Locate is a new reserved operation name to use in request messages, the expected response to which is a reply with a LOCATION_FORWARD status value and an encoded IOR in the message body or an Object reference return value with a normal success return status.

A transport connection opened with the endpoint information of a ForwardingServiceLocator can be used to locate ObjectGroups by despatching CORBA requests to it as though the objects were located there. The behaviour of the ForwardingServiceLocator endpoint when it receives a request for an object_key that would normally not exist there is the following:

1. Utilise whatever ***RequestDecoder***s it has registered to extract an ObjectGroupName.
2. If not successful then the ForwardingServiceLocator will return OBJECT_NOT_EXIST system exception with COMPLETED_NO completion status.
3. If an ObjectGroupName is decoded then the value will be passed to Locator::locate
4. If the locate call returns an ObjectGroup the behaviour depends upon the request_id of the request. If the request id was FT_Locate then the ObjectGroup should be returned to the client. Any other request to the ObjectGroup should be returned as a LOCATION_FORWARD response.

9.37 RequestDecoder

A request decoder provides a portable way to provide a mechanism to extract the ObjectGroupName from a CORBA Request. These can be plugged in at any PI request interception point to provide ObjectGroup location capabilities. It is envisaged that ORB / service implementers will provide a mechanism for application developers to register an ordered list of these to provide location capabilities (see ServiceLocator interface).

```
module LWFT
{
    local interface RequestDecoder
    {
        // From the given request extract the ObjectGroupName
        boolean get_name_from_request
            (in PortableInterceptor::RequestInfo request,
            out ObjectGroupName group_name);
    };
};
```

9.38 Necessary ORB modifications

Fully compliant ORBs must be capable of accepting and correctly responding to GIOP request messages using a new reserved operation name FT_Locate. In this section we will explain why there is a need for this. The details of how ORBs that do not implement this message type can still interact with the service will be covered in §9.39.

The key principle behind the correct functioning of the LWFT mechanism proposed here is the notion of forwarding references between a Client and a suitable Server replica. Some aspects of the RT CORBA specification require that information regarding individual objects be provided in the IOR. To support this, while it is not required that all objects be registered at the point that they are created, it is necessary for the Forwarder to be capable of querying a particular object for it's full direct (i.e. not redirected via the Forwarder) IOR. The LWFT mechanism therefore requires that the response given to any request message is a LOCATION_FORWARD one. This differs to the standard GIOP LocateRequest message type.

As defined in Chapter 15 of the CORBA 3.0.3 Specification ('General Inter-ORB Protocol'), there are two possible request message types which can be used by standard ORBs: Request and LocateRequest. The standard GIOP response types for these request types are Reply and LocateReply respectively. Provided that no exception occurs there are two valid status values that can be carried in the header of a Reply message, and three in a LocateReply message:

Request Type	Response Type	Possible non-exceptional header status values	Meaning
Request	Reply	NO_EXCEPTION	The message body contains the result of an operation invocation.
		LOCATION_FORWARD	The message body contains an IOR.
LocateRequest	LocateReply	OBJECT_HERE	The server returning this message can accept requests for the object specified in the LocateRequest message.
		OBJECT_FORWARD	The message body contains an IOR.
		UNKNOWN_OBJECT	The object specified in the LocateRequest message is not known at this server.

Table 2: The GIOP response types and their possible status header values

Both Reply and LocateReply have a status header value which enables them to forward Clients IORs in their message body. The problem with LocateReply is that the CORBA specification does not dictate that all ORBs must implement location forwarding behaviour at the Server's side. ORBs which do not implement location forwarding will therefore only ever respond to LocateRequest messages using OBJECT_HERE or UNKNOWN_OBJECT status values. There may also be ORB implementations which do not always choose to return forwarding type responses to requests.

This is clearly a problem for object group references created by a POA with ObjectGroupMembershipPolicy value of AUTO_ORB_CTRL as it relies entirely on the status of a response message being a reference forwarding type every time. The solution to this problem is to implement a new reserved request operation, FT_Locate, to which the only possible (non-exceptional) response is either a LOCATION_FORWARD or a normal return of an object reference.

9.39 Providing LWFT to unmodified ORBs

An important requirement of this LWFT design is that it can still provide a limited service to ORBs that do not implement the new FT_Locate reserved operation name described above in §9.36. If a Server ORB with POAs using ObjectGroupMembershipPolicy value of AUTO_ORB_CTRL does not support requests using FT_Locate, it is not possible for the Forwarder to obtain an IOR for an individual object from the Server. As a result of this, any IOR which is forwarded to the Client (the CORBA standards state that any ORB must be able to process a LOCATION_FORWARD on the Client's side) must come from the Forwarder alone. In this situation it is possible for the Forwarder to generate an IOR for the Client using the endpoint information that it usually uses to contact the Server. Forwarders should interpret a BAD_OPERATION exception response to a request using FT_Locate as indicating that a Server replica object does exist, but that the Server ORB does not support this reserved operation name. It will be possible to control which Server replicas will be contacted using the FT_Locate operation name by using a suitable policy.

While it may initially sound like there are no negative effects produced by applications using the LWFT Middleware in this manner, there unfortunately is one key problem: any application not using location forwarding on the Server's side will not be fully compatible with RT CORBA or any other services which require the use of IOR Tagged Components on a per object (rather than per ORB or per POA) basis. In the normal approach, the IOR forwarded back from the Server can contain additional pieces of information, in the form of IOR Tagged Components, which are required to provide protocol support for applications implementing the Real-time CORBA specification. For Servers not implementing location forwarding the Forwarder must build an IOR using the information that the Server registered itself with. As these registered details might contain out of date tagged information, if any at all, the IORs generated from this data will not be suitable for use by Real-time applications.

Although Server ORBs not implementing FT_Locate will not be fully compatible with RT CORBA, or any other service requiring the presence of up to date additional information in IORs, they will still be able to use the basic LWFT service. This ensures that unmodified ORBs will be able to gain some advantage from the proposed LWFT service, and that as high a level of interoperability as possible is provided.