Fault Tolerant CORBA Specification, V1.0

OMG document: ptc/2000-04-04
replaces draft adopted specification ptc/2000-03-04 and submission document orbos/00-01-19

This document is an OMG Final Adopted Specification, which has been approved by the OMG board and technical plenaries, and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by July 21, 2000.

You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*; however, at the time of this writing there were no pending issues.

The FTF Recommendation and Report for this specification will be published on December 15, 2000. If you are reading this after that date, please download the available specification from the OMG formal specifications web page *http://www.omg.org/library/specindx.html*.

OMG Adopted Specification

Fault Tolerant CORBA Specification 27

Note – The OMG documents used to create this chapter were orbos/00-01-19 and ptc/2000-03-04.

Contents

This chapter contains the following topics.

27.1 Fault Tolerant CORBA

27.1.1 Fault Tolerance for Diverse Applications

Many different kinds of applications, developed by the members of the OMG and the users of CORBA, have a need for fault tolerance. These applications range from very large critical systems (such as air traffic control and defense systems) to smaller critical systems (such as 911 and medical systems) to embedded applications (such as aircraft instrumentation and manufacturing control applications) to communication systems (such as telephony and networking systems) to enterprise applications (such as financial and supply chain applications).

A standard that attempts to meet all of the requirements of this wide spectrum of applications might satisfy many needs only poorly, or might be too complex to implement. This specification therefore represents a number of compromises. In particular, to provide full interoperability between the products of different vendors, substantially more interfaces and protocols would need to be defined than are defined in this specification. Once experience of implementation and use of the proposed specification has been gained, it might be appropriate to extend the specification to provide greater interoperability and fault tolerance. In the meantime, some vendors may choose to offer proprietary extensions to satisfy the fault tolerance needs of specific kinds of applications.

27.1.2 Objectives

The standard for Fault Tolerant CORBA aims to provide robust support for applications that require a high level of reliability, including applications that require more reliability than can be provided by a single backup server. The standard requires that there shall be no single point of failure.

Fault tolerance depends on entity redundancy, fault detection and recovery. The entity redundancy by which this specification provides fault tolerance is the replication of objects. This strategy allows greater flexibility in configuration management of the number of replicas, and of their assignment to different hosts, compared to server replication. Replicated objects can invoke the methods of other replicated objects without regard to the physical location of those objects. Support for redundancy in time is provided by allowing clients to make repeated requests on the server, using the same or alternative transport paths.

The standard supports a range of fault tolerance strategies, including request retry, redirection to an alternative server, passive (primary/backup) replication, and active replication which provides more rapid recovery from faults. The standard allows the users to define fault tolerance properties for each replicated object (object group).

The standard supports applications that require the Fault Tolerance Infrastructure to control the creation of the application object replicas, as well as applications that control directly the creation of their own object replicas. It supports applications that require the Fault Tolerance Infrastructure to maintain Strong Replica Consistency, both under normal conditions and also under fault conditions, as well as applications that provide whatever level of consistency they require.

The standard provides support for fault detection, notification, and analysis for the object replicas. It supports applications that require the Fault Tolerance Infrastructure to provide automatic checkpointing, logging and recovery from faults, as well as applications that handle their own fault recovery.

The standard aims for minimal modifications to the application programs, and for transparency to replication and to faults. It defines minimal modifications to existing ORBs that allow non-replicated clients to derive fault tolerance benefits when they invoke replicated server objects.

27.1.3 Basic Concepts

27.1.3.1 Replication and Object Groups

To render an object fault-tolerant, several replicas of the object are created and managed as an *object group*. While each individual replica of an object has its own object reference, an additional *interoperable object group reference* (IOGR) is introduced for the object group as a whole. It is this object group reference that the replicated server publishes for use by the client objects. The client objects invoke methods on the server object group, and the members of the server object group execute the methods and return their responses to the clients, just like a conventional object. Because of the object group abstraction, the client objects are not aware that the server objects are replicated (*replication transparency*) and are not aware of faults in the server replicas or of recovery from faults (*failure transparency*).

27.1.3.2 Fault Tolerance Domains

Many applications that need fault tolerance are quite large and complex. Managing such an application as a single entity is inappropriate. Consequently, this specification defines *fault tolerance domains*, as illustrated in Figure 27-1 on page 27-4. Each fault tolerance domain typically contains several hosts and many object groups, and a single host may support several fault tolerance domains. Existing security policies and mechanisms can be maintained by ensuring that a fault tolerance domain is entirely contained within a single security domain. All of the objects groups within a fault tolerance domain are created and managed by a single Replication Manager, but they can invoke and can be invoked by objects within other fault tolerance domains. The concept of fault tolerance domains allows applications to scale to arbitrary sizes, by allowing a smaller number of objects to be managed by each Replication Manager.

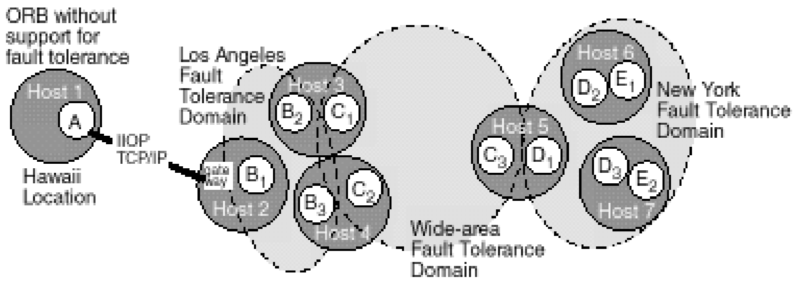
ORB without support for fault tolerance
Host 1
A
Hawaii Location
IIOP TCP/IP
Los Angeles Fault Tolerance Domain
Host 3
B_2
C_1
Gateway
B_1
Host 2
B_3
C_2
Host 4
Host 5
C_3
D_1
Wide-area Fault Tolerance Domain
Host 6
D_2
E_1
New York Fault Tolerance Domain
D_3
E_2
Host 7

Figure 27-1 Fault tolerance domains are shown lightly shaded, hosts are shown darkly shaded, and members of an object group are shown unshaded. The members of object group B are denoted B1, B2 and B3, and similarly for object groups C, D and E.

27.1.3.3 *Fault Tolerance Properties*

Each object group has an associated set of fault tolerance properties. Examples of such properties are the **ReplicationStyle** (COLD_PASSIVE, WARM_PASSIVE, ACTIVE, etc), **InitialNumberReplicas**, **MinimumNumberReplicas**, etc. It is possible to define fault tolerance properties that apply to all object groups within a fault tolerance domain or to all object groups of a specific type. It is also possible to set the properties of an object group when it is created, and to change the properties dynamically after the object group is created.

27.1.3.4 *Strong Replica Consistency*

Strong replica consistency requires that the states of the members of an object group remain consistent (identical) as methods are invoked on the object group and as faults occur. More specifically, for the ACTIVE **ReplicationStyle**, Strong Replica Consistency means that, at the end of each method invocation on the object group, all of the members of the object group have the same state. For the COLD_PASSIVE and WARM_PASSIVE **ReplicationStyle**s, it means that, at the end of each state transfer, all of the members of the object group have the same state. Strong Replica Consistency requires Strong Group Membership, as well as Uniqueness of the Primary for passive replication. Strong Group Membership means that, for each method invocation on an object group, the Fault Tolerance Infrastructures on all hosts have the same view of the membership of the object group. Uniqueness of the Primary for passive replication means that one and only one member of the object group executes the methods invoked on the object group.

27.1.4 Architectural Overview

Figure 27-2 presents an architectural overview of a fault-tolerant system, showing an example strategy for implementation of the specifications for Fault Tolerant CORBA. Other implementation strategies are possible.

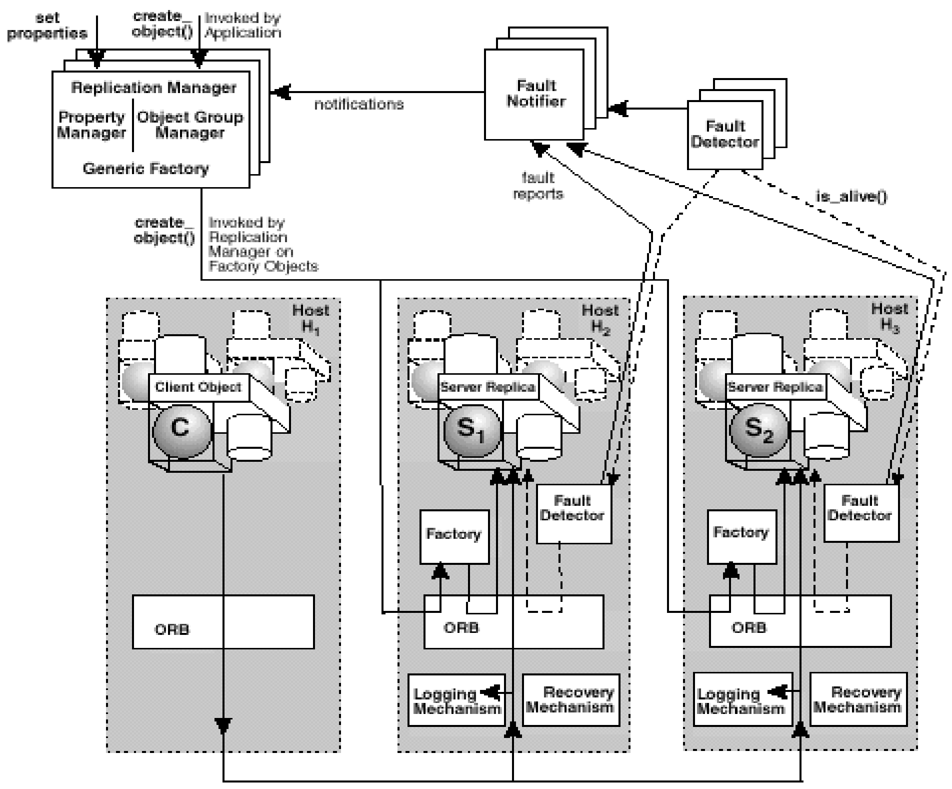

Figure 27-2 An Achitectural Overview of a Fault-Tolerant System.

At the top of the figure are shown several components of the Fault Tolerance Infrastructure (Replication Manager, Fault Notifier, Fault Detector), all of which are implemented as CORBA objects. Logically, there is a single instance of the Replication Manager and Fault Notifier in each fault tolerance domain but, physically, they are replicated to protect against faults, just as are the application objects. The Replication Manager inherits the **PropertyManager**, **ObjectGroupManager**, and **GenericFactory** interfaces.

The bottom of the figure shows three hosts, as follows:

- a client application object C on host H_1 that is invoking a replicated server object with two replicas,

- S_1 on host H_2 and
- S_2 on host H_3.

A typical system will contain many such client and server objects.

The figure shows Factory and Fault Detector objects that may be present on each host and are specific to that host. These host-specific objects are not replicated, unlike the service objects shown at the top of the figure, which are replicated objects.

The figure also shows the Message Handler and the Logging and Recovery Mechanisms that are present on each host. These are not CORBA objects but, rather, are a part of the ORB, or are located between the ORB and the operating system.

27.1.4.1 Fault Tolerance Property Management

This specification provides a **PropertyManager** interface that allows the user to define fault tolerance properties of object groups. The specification of the **PropertyManager** interface is designed to allow vendors to develop graphical user interfaces and to define additional properties should they so desire.

Two properties of particular relevance are the Membership Style and the Consistency Style. The Membership Style defines whether the membership of an object group is infrastructure-controlled or application-controlled. Similarly, the Consistency Style defines whether the consistency of the states of the members of an object group is infrastructure-controlled or application-controlled. Some components of the Fault Tolerance Infrastructure, such as the Logging and Recovery Mechanisms, are used only for object groups that have the infrastructure-controlled Consistency Style.

27.1.4.2 Replication Management

For the infrastructure-controlled (MEMB_INF_CTRL) Membership Style (Section 27.3.2.2, “MembershipStyle,” on page 27-32) the replication of objects is substantially transparent to the application program, which simplifies the development of new application programs, and allows the continued use of existing application programs.

Using the **create_object()** method of the **GenericFactory** interface, the application program requests the creation of a replicated object (object group), just as it would an unreplicated object. This method is invoked on the Replication Manager, rather than directly on the factory (as it would have been in the unreplicated case). The Replication Manager then invokes the factories, on the different hosts, where a replica is to be created, using the same **create_object()** method of the **GenericFactory** interface.

Using the **create_member(), add_member()**, and **remove_member()** methods of the **ObjectGroupManager** interface, the application can exercise control over the addition and removal, and location, of members of an object group (violating transparency).

While each individual replica has its own object reference, the object group as a whole has its interoperable object group reference, which is created by the Replication Manager. This object group reference contains a TAG_FT_GROUP component for the object group within the profiles of the object group reference. The object group reference is returned to the application by the Replication Manager, and is published by the server object. The client objects use the object group reference to invoke methods on the server object group, just as they would have used a conventional object reference for an unreplicated object.

Because of the object group abstraction, the client objects are not aware that the server objects are replicated (client transparency to replication), and are not aware of faults in the server replicas or of the recovery of server replicas when a fault has occurred (client transparency to faults).

27.1.4.3 *Fault Detection and Notification*

Fault tolerance requires fault detection, and typical systems contain several fault detection mechanisms to detect host failures, resource exhaustion, etc. This specification defines a simple **PullMonitorable** interface that the application objects inherit. The **PullMonitorable** interface contains the **is_alive()** method that a Fault Detector invokes. For efficiency, the Fault Detector that monitors an application object is typically located on the same host as that object, while the local Fault Detectors are monitored by a global Fault Detector that is replicated for fault tolerance.

The Fault Detector, and also other kinds of fault detectors in the system, such as those based on the PUSH Monitoring Style and those that detect host or network faults, report faults to the Fault Notifier, which passes fault notifications to the Replication Manager and other objects that have registered for such notifications. An application-specific fault analyzer may register to receive such notifications, and may condense and filter such notifications into further fault reports that it returns to the Fault Notifier.

27.1.4.4 *Logging and Recovery*

For the COLD_PASSIVE and WARM_PASSIVE Replication Styles, under fault-free conditions, only one member of an object group, the primary member, executes the requests and generates the replies. If the Fault Detector suspects that the primary member is faulty, the Replication Manager, at its discretion, restarts the current primary member or promotes a backup member to become the new primary member.

For the application-controlled (CONS_APP_CTRL) Consistency Style, the Replication Manager takes no further recovery action and the new primary member is responsible for the recovery of its own state.

For the infrastructure-controlled (CONS_INF_CTRL) Consistency Style, the new primary member must start operation with the appropriate state, and must execute the same sequence of requests that were, or should have been, executed by the previous primary member, had it not failed. Thus, each GIOP message is passed to the Logging and Recovery Mechanisms, automatically and invisibly to the application. The Logging Mechanism records the message in a log, from which the Recovery Mechanism can retrieve the message during recovery.

Periodically, the Logging Mechanism invokes the **get_state()** method of the **Checkpointable** interface, which must be implemented by every replicated application object, to obtain the state of the object, so that the state can be recorded in a log. During recovery, the Recovery Mechanism invokes the **set_state()** method of the **Checkpointable** interface of the new primary to set its state to the state that was recorded in the log.

27.1.5 Requirements

The requirements of the Fault Tolerant CORBA specification are stated below.

CORBA Object Model

For object groups with the infrastructure-controlled (CONS_INF_CTRL) Consistency Style (Section 27.3.2.3, “ConsistencyStyle,” on page 27-33), the specification requires that the CORBA object model is preserved. Even though an object is replicated to provide protection against faults, at all times its behavior shall appear to be the behavior of a single object. In particular, a replicated object can act as a client or a server or both, and can invoke another replicated object, regardless of the fault tolerance properties of the two object groups.

CORBA Object Reference Model

The specification introduces three new special tagged components into the CORBA object reference model. The object group references that are used for fault tolerance contain multiple profiles that contain these components. Even though an object group reference contains such components in its profiles, an unreplicated object, hosted by an ORB that does not support fault tolerance, can still use the reference to invoke the methods of the replicated object. Similarly, a replicated object can use the object reference of an unreplicated object to invoke the methods of the unreplicated object.

Transparency to Replication and to Faults

Creating or deleting an object using a Generic Factory, and invoking a method of an object, appear the same for replicated objects as for unreplicated objects. Similarly, the behavior of a replicated server object when invoked by a client object appears the same whether or not faults occur, except perhaps for a transient delay if the primary member of a passively replicated object becomes faulty.

No Single Point of Failure

The specification supports applications that need robust fault tolerance, including applications that require higher reliability than can be provided by a single backup. The specification requires that there shall be no single points of failure.

Client Redirection

For a client and a replicated server, the specification defines an interoperable object group reference that allows the client to connect to the server replicas, by connecting to an alternative server or through an alternative network, when a fault in a server replica occurs. It defines an additional service context, in request messages, that allows a server to determine if the object group reference for the server used by a client is obsolete. Transparency to the client application program is provided, with minimal modifications to the client ORB and simple mechanisms in the server ORB. Typical applications include desktop client access to enterprise servers.

Transparent Reinvocation

The specification introduces an additional service context in Request messages that ensures that, in the presence of faults, a client can reinvoke a request on a replicated server and receive a reply to that request, without risk that the operation will be performed more than once. Typical applications include desktop client access to e-commerce applications.

Infrastructure-Controlled Membership

The infrastructure-controlled (MEMB_INF_CTRL) Membership Style allows the application to direct the Replication Manager to create an object group. The Replication Manager then invokes the factories at the different locations to create the object replicas, and then add them to the group. The Replication Manager is responsible for creating the initial number of replicas and for maintaining the minimum number of replicas, as specified by the fault tolerance properties for the group. Typical applications include enterprise server applications, such as supply chain applications, and large-scale critical systems, such as defense applications.

Application-Controlled Membership

The application-controlled (MEMB_INF_CTRL) Membership Style allows the application to create the members of an object group and to direct the Replication Manager to add them to the group, or to direct the Replication Manager to create the members of an object group and add them to the group. The application is responsible for maintaining the initial and minimum number of replicas and the locations of the replicas, both initially and also after faults. Application-controlled membership is particularly important for applications whose different hosts have different capabilities, such as communication network applications.

Infrastructure-Controlled Consistency

The infrastructure-controlled (CONS_INF_CTRL) Consistency Style provides Strong Replica Consistency between the states of the members of an object group. Strong Replica Consistency requires that, even in the presence of faults, as members of an object group execute a sequence of methods invoked on the object group, the behavior is logically equivalent to that of a single fault-free object processing the same sequence of method invocations. The Fault Tolerance Infrastructure provides logging, checkpointing, activation and recovery mechanisms to achieve Strong Replica

Consistency. Strong Replica Consistency is particularly important for financial applications and safety-critical applications, such as industrial process control and aircraft instrumentation.

Application-Controlled Consistency

The application-controlled (CONS_APP_CTRL) Consistency Style depends on application-specific mechanisms to ensure whatever consistency is required for the members of an object group. Application-controlled consistency does not depend on the Fault Tolerance Infrastructure to provide logging, checkpointing or recovery, and does not guarantee Strong Replica Consistency. Typical applications might include telecommunications applications, and also some embedded and real-time applications.

Passive Replication

The COLD_PASSIVE or WARM_PASSIVE ReplicationStyles require that, during fault-free operation, only one member of the object group, the primary member, executes the methods invoked on the group. Periodically, the state of the primary member is recorded in a log, together with the sequence of method invocations. In the presence of a fault, a backup member is promoted to be the new primary member of the group. The state of the new primary is restored to the state of the old primary by reloading its state from the log, followed by reapplying request messages recorded in the log. Passive replication is useful when the cost of executing a method invocation is larger than the cost of transferring a state, and the time for recovery after a fault is not constrained. Typical examples include enterprise inventory, logistics applications and hospital record keeping.

Active Replication

The ACTIVE ReplicationStyle requires that all of the members of an object group execute each invocation independently but in the same order, so that they maintain exactly the same state and, in the event of a fault in one member, that the application can continue with results from another member without waiting for fault detection and recovery. Even though each of the members of the object group generates each request and each reply, the Message Handling Mechanism detects and suppresses duplicate requests and replies, and delivers a single request or reply to the destination object(s). Active replication is useful when the cost of transferring a state is larger than the cost of executing a method invocation, or when the time available for recovery after a fault is tightly constrained. Typical examples include enterprise electronic trading applications and safety-critical applications, such as hospital patient monitoring.

Fault Detection and Notification

The Fault Management interfaces allow detection of object crash faults, and provide fault notifications to the entities that have registered for such notifications. Accuracy of fault detection is impossible in an asynchronous fault-tolerant distributed system. Occasional false suspicions cause no harm in a robust fault-tolerant system. If a host crashes or an object hangs, the Fault Detectors are required to detect the fault in a timely manner. However, a Fault Detector must not continuously suspect all members of an object group, unless all of them are indeed faulty. Most fault-tolerant applications will use the Fault Management interfaces, but they are particularly important for telecommunications, electric power distribution and other safety-critical applications.

Logging and Recovery

The Logging and Recovery Mechanisms and **Checkpointable** and **Updateable** interfaces allow an application object to record its state, for use in recovery after a fault or to initialize another replica. Following a fault that damages one or more, but not all, of the members of an object group, recovery is required to ensure that the continued behavior of the replicated object after recovery is the same as it would have been in the absence of the fault. A recovering member executes the same requests in the same order, generates the same replies, invokes the same methods of other objects, and reaches the same internal state, as if no fault had occurred. If a request is partially executed when a fault occurs, that request is fully executed, at the same position in the seqence of messages, during recovery. If an object invokes a method of another object and then becomes faulty, that method invocation must not be duplicated during recovery. Because some objects may be unreplicated, or may be supported by ORBs that do not provide fault tolerance, or may use different Replication Styles, the recovery of each object must be self-contained and must not depend on the cooperation of any other object. Applications that employ the infrastructure-controlled Consistency Style will use these mechanisms and interfaces.

27.1.6 Limitations

The limitations of the Fault Tolerant CORBA specification are given below.

Legacy ORBs

An unreplicated client hosted by a legacy ORB can invoke methods of a replicated server, supported by the Fault Tolerance Infrastructure. The object group references generated for replicated servers can be used by legacy ORBs, although the full benefits of fault-tolerant operation are not achieved for an unreplicated client. If a legacy ORB has been modified to understand object group references and to retry requests at alternative destinations, the unreplicated client receives the benefits of a higher, but still partial, level of fault tolerance. Special service contexts in the request and reply messages protect an unreplicated client from a replicated server executing its requests multiple times when the client retries those requests at alternative destinations.

Common Infrastructure

All of the hosts within a fault tolerance domain must use ORBs from the same vendor and Fault Tolerance Infrastructures from the same vendor to ensure interoperability and full fault tolerance within that domain. Consequently, the members of an object group must be hosted by ORBs from the same vendor and Fault Tolerance Infrastructures from the same vendor. For clients and servers in different fault tolerance domains, both using ORBs and Fault Tolerance Infrastructures from the same vendors, full fault tolerance can be achieved. Otherwise, the specifications provide a useful improvement over no fault tolerance but substantially less than full fault tolerance.

Deterministic Behavior

For the infrastructure-controlled Consistency Style, for both active and passive replication, deterministic behavior is required of the application objects, and also of the ORBs, to guarantee Strong Replica Consistency. The inputs to the replicas of an object must be consistent (identical); this implies that request and reply messages must be delivered in the same order to each of the replicas of an object. If sources of non-determinism exist, they must be filtered out. Multi-threading in the application or the ORB may be restricted, or transactional abort/rollback mechanisms may be used.

Network Partitioning Faults

Network partitioning faults separate the hosts of the system into two or more sets, the hosts of each set being able to operate and to communicate within that set but not with hosts of different sets. The current state-of-the-art does not provide an adequate solution to network partitioning faults. Thus, network partitioning faults are not addressed in this specification.

Commission Faults

A commission fault occurs when an object or host generates incorrect results. A Byzantine fault is a commission fault in which an object or host generates incorrect results maliciously. Algorithms have been devised to detect and protect against a fairly wide range of Byzantine faults but they are complex and expensive in processing and communication. In the current state-of-the-art, Byzantine algorithms are seldom appropriate for fault tolerance but might be appropriate for security, to protect a system after one or more of its hosts have been subverted by intruders. The specification provides an ACTIVE_WITH_VOTING Replication Style. Voting itself is relatively inexpensive, but the communications infrastructure required to support voting properly is substantially more expensive than that required to tolerate only crash faults.

Correlated Faults

No protection is provided against design or programming faults, or other correlated faults, that cause the same errors in all replicas of an object, in all ORBs, or in all hosts or their operating systems.

27.2 Basic Fault Tolerance Mechanisms

27.2.1 Overview

This section defines basic fault tolerance mechanisms that must be implemented for Fault Tolerant CORBA. The client-side mechanisms are intended to be simple light-weight extensions to CORBA that will be easy to implement. These mechanisms enable client-side ORBs to achieve a higher level of reliability by exploiting the fault tolerance mechanisms defined for server-side ORBs.

In particular, this section defines:

- Interoperable object group reference that contains multiple **TAG_INTERNET_IOP** profiles, each of which contains the **TAG_FT_GROUP** component and one of which may contain a **TAG_FT_PRIMARY** component. The interoperable object group reference may contain the **TAG_MULTIPLE_COMPONENTS** profile, which may contain the **TAG_FT_GROUP** component.
- Failover semantics for Fault Tolerant CORBA that extend the failover semantics for the CORBA core.
- Most recent object group reference for a server object group, using the **FT_GROUP_VERSION** service context in a client's request message. The **FT_GROUP_VERSION** service context allows the server to determine whether the client is using the most recent object group reference for the server object group.
- Transparent reinvocations of requests, using the **FT_REQUEST** service context in a client's request messages, the client-side Request Duration Policy and the fault handling semantics of GIOP messages. The **FT_REQUEST** service context prevents a request from being executed two or more times as a consequence of reinvocation of the request on a backup server after a fault.
- Heartbeating of the server, using the **TAG_FT_HEARTBEAT_ENABLED** component of the **TAG_INTERNET_IOP** profile, the client-side Heartbeat Policy and the server-side Heartbeat Enabled Policy. This allows the client to detect failure of the server.

27.2.2 Interoperable Object Group References

This section extends the definition of an interoperable object reference (IOR) to encompass references to server object groups. The interoperable object group reference (IOGR) for a server object group is an IOR that contains multiple **TAG_INTERNET_IOP** profiles and that may contain a **TAG_MULTIPLE_COMPONENTS** profile.

Each of the **TAG_INTERNET_IOP** profiles must contain the **TAG_FT_GROUP** component, and may contain other components such as **TAG_IIOP_ALTERNATE_ADDRESS** components. At most one of the **TAG_INTERNET_IOP** profiles may contain the **TAG_FT_PRIMARY** component. The **TAG_MULTIPLE_COMPONENTS** profile may also contain the **TAG_FT_GROUP** component, which must be used for object groups that have no members. An example is shown in Figure 27-3 on page 27-14.

The **TAG_FT_GROUP** component and **TAG_FT_PRIMARY** component are described in Section 27.2.2.1, "TAG_FT_GROUP Component," on page 27-14 and Section 27.2.2.2, "TAG_FT_PRIMARY Component," on page 27-16.

.

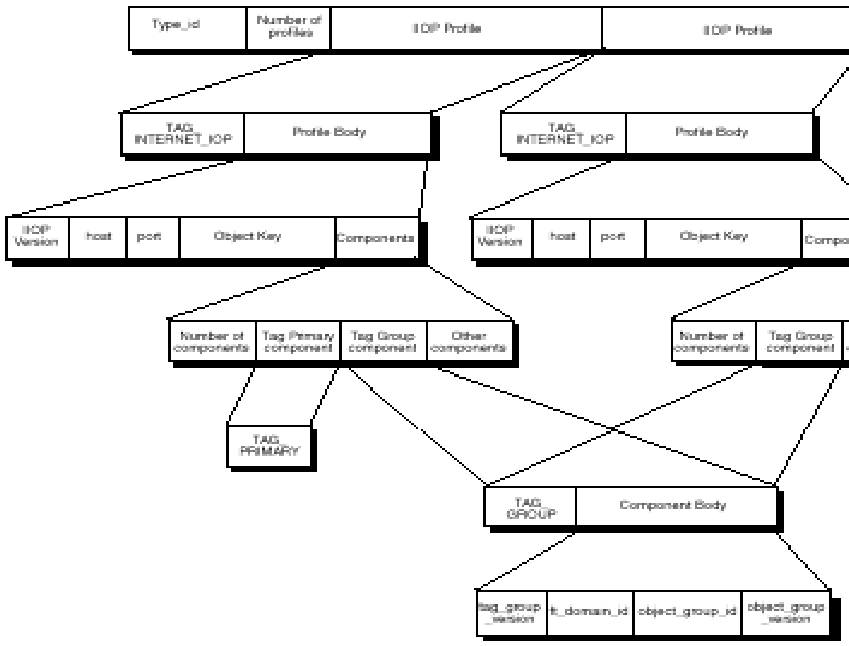

Figure 27-3 An example of the Interoperable Object Group Reference Used for Fault Tolerance. The reference is an IOR that contains multiple TAG_INTERNET_IOP profiles, any of which may be used to reach the server object group. The reference may also contain a TAG_MULTIPLE_COMPONENTS profile. The TAG_FT_GROUP component is contained in every profile of the reference. The TAG_FT_PRIMARY component is contained in at most one TAG_INTERNET_IOP profile.

27.2.2.1 *TAG_FT_GROUP Component*

The **TAG_FT_GROUP** component is contained in the profiles of the interoperable object group reference.

```
module IOP {
    const ComponentId TAG_FT_GROUP = 27;
};

module FT {
    typedef string FTDomainId;
    typedef unsigned long long ObjectGroupId;
    typedef unsigned long ObjectGroupRefVersion;

    struct TagFTGroupTaggedComponent { // tag = TAG_FT_GROUP;
            GIOP::Version                   version;
            FTDomainId                      ft_domain_id;
            ObjectGroupId                   object_group_id;
            ObjectGroupRefVersion           object_group_ref_version;
    };
};
```

Object groups have an identity that persists even as the membership of the object group changes. Thus, an object group requires an identifier that is unique within the context of a fault tolerance domain. Moreover, as the membership of an object group changes, the object group reference may have different versions. To address these concerns, Fault Tolerant CORBA introduces the following types.

typedef string FTDomainId;

The identifier of a fault tolerance domain.

typedef unsigned long long ObjectGroupId;

The identifier of an object group.

typedef unsigned long ObjectGroupRefVersion;

The version number of the object group reference.

The **TAG_FT_GROUP** component contains the fault tolerance domain identifier and object group identifier of the server object group, which are used to reach the server object group. It also contains the object_group_ref_version, which the client ORB may put in the **FT_GROUP_VERSION** service context in the client's request messages, as described in Section 27.2.7.1, "FT_GROUP_VERSION Service Context," on page 27-22.

const ComponentId TAG_FT_GROUP = 27;

A constant that designates the **TAG_FT_GROUP** component that is contained in the **TAG_INTERNET_IOP** profiles and also may be contained in the **TAG_MULTIPLE_COMPONENTS** profile.

```
struct TagFTGroupTaggedComponent { // tag = TAG_FT_GROUP;
GIOP::Version                version;
FTDomainId                   ft_domain_id;
ObjectGroupId                object_group_id;
ObjectGroupRefVersion      object_group_ref_version;
};
```

The **TAG_FT_GROUP** component, within the **TAG_INTERNET_IOP** profiles and **TAG_MULTIPLE_COMPONENTS** profile, contains the version of the **TAG_FT_GROUP** component, the fault tolerance domain identifier, the object group identifier, and the version number of the object group reference for the server object group. For implementations conforming to this version of the specification, the value of version.major must be 1 and the value of the version.minor must be 0.

27.2.2.2 TAG_FT_PRIMARY Component

The **TAG_FT_PRIMARY** component is contained in at most one of the **TAG_INTERNET_IOP** profiles of the interoperable object group reference.

```
module IOP {
        const ComponentId TAG_FT_PRIMARY = 28;
};

module FT {
        struct TagFTPrimaryTaggedComponent { // tag = TAG_FT_PRIMARY;
                boolean primary;
        };
};
```

The profile that contains the **TAG_FT_PRIMARY** component is used in preference to other profiles to reach the server object group.

const ComponentId TAG_FT_PRIMARY = 28;

A constant that designates the **TAG_FT_PRIMARY** component that is contained in at most one of the **TAG_INTERNET_IOP** profiles.

```
struct TagFTPrimaryTaggedComponent { // tag = TAG_FT_PRIMARY;
        boolean primary;
};
```

The **TagFTPrimaryTaggedComponent**, when present in a **TAG_INTERNET_IOP** profile, indicates that the profile is to be used in preference to the other **TAG_INTERNET_IOP** profiles within the object group reference.

At most one of the profiles in the object group reference contains the **TAG_FT_PRIMARY** component. A client-side ORB may use that profile in preference to the other profiles. It is not mandated that the ORB must choose the profile containing the **TAG_FT_PRIMARY** component. Moreover, it cannot be guaranteed that the endpoint addressed by the profile containing the **TAG_FT_PRIMARY** component is currently the primary endpoint for the object group.

Use of any of the profiles, other than that containing the **TAG_FT_PRIMARY** component, may result in one or more **LOCATION_FORWARD**s and thus reduced efficiency. No requirement is imposed on the particular order in which the other profiles, that do not contain the **TAG_FT_PRIMARY** component, must be used.

27.2.3 Interoperable Object Group Reference Operations

IOGRs are IORs. However, the semantics of several of the operations inherited from **CORBA::Object** must be adjusted to account for the group contents of an IOGR.

27.2.3.1 get_interface

Unchanged. The assembly procedure for an object group guarantees that the interfaces supported by the object group are supported by all members of the object group.

27.2.3.2 *is_a*

Unchanged.

27.2.3.3 *is_nil*

Essentially unchanged. True if no profiles are present or if **is_nil** is true for all of the profiles.

27.2.3.4 *non_existent*

Essentially unchanged. True if the object group does not exist. Note that the object group might exist even if **non_existent()** is true for all of the profiles of the object group reference or even if there are no IOP profiles in the object group reference. (This occurs when an object group with the application-controlled Membership Style is created with no members so that the members can be added individually by the application.) A server ORB can obtain an authoritative determination of non-existence of the object group from the Replication Manager, using the same mechamisms as are used to obtain the most recent object group reference. The ORB must use those mechanisms to generate a **LOCATION_FORWARD** reply when the client's request contains an obsolete **object_group_ref_version** field in the **FT_GROUP_VERSION** service context.

27.2.3.5 *is_equivalent*

There are three cases to consider for checking equivalence:

1. Two non-object group references. The semantics of the operation are unchanged in this case.
2. An object group reference and a non-object group reference. These references are not equivalent.
3. Two object group references. Section 27.2.2.1, "TAG_FT_GROUP Component," on page 27-14 introduces a strong identity for an object group in its **ft_domain_id** and **object_group_id** fields. Two object group references are equivalent if they have the same **ft_domain_id** and the same **object_group_id** fields. Note that the **object_group_ref_version** field in the **TAG_FT_GROUP** component is ignored.

The analysis of these cases collapses the semantics to the following:

- Non-Fault-Tolerant CORBA implementations are essentially unchanged. These implementations might not recognize certain object group references as representing the same object group. However, that is allowed under the present semantics.
- Fault Tolerant CORBA implementations compare the values of the corresponding **ft_domain_id** and **object_group_id** fields in the **TAG_FT_GROUP** components to determine the equivalence of two object group references. Otherwise, the semantics for **is_equivalent** are unchanged.

27.2.3.6 *hash*

Follows the semantics above.

27.2.3.7 *create_request*

Unchanged.

27.2.3.8 *get_policy*

Unchanged.

27.2.3.9 *get_domain_managers*

Unchanged.

27.2.3.10 *set_policy_overrides*

Unchanged.

27.2.4 *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).

The choice of addressing mode is influenced by the Replication Style of the object group.

27.2.4.1 *Profiles That Address Object Group Members*

When using profiles that address members of an object group, the object group reference for a server object group contains one **TAG_INTERNET_IOP** profile for each member of that group. Each profile contains a member reference that can be used to reach an individual member of the object group.

27.2.4.2 *Profiles That Address Gateways*

When using profiles that address gateways, the object group reference for a server object group contains one **TAG_INTERNET_IOP** profile for each of several alternative gateways to that group. Each profile contains a reference to a gateway that can forward messages to all members of the server object group possibly using a proprietary multicast group communication protocol. The group communication protocol may be used for server object groups that support any of the Replication Styles.

27.2.4.3 *Choice of Profile Addressing Mode*

For a server object group having the **STATELESS**, **COLD_PASSIVE**, or **WARM_PASSIVE** ReplicationStyles (Section 27.3, "Replication Management," on page 27-30), the Fault Tolerance Infrastructure at the server may create either an object group reference that contains member profiles, or alternatively, an object group reference that contains gateway profiles.

For a server object group having the **ACTIVE** and **ACTIVE_WITH_VOTING** (Section 27.3.2, "Fault Tolerance Properties," on page 27-31) Replication Styles, the client must invoke all of the members of the server object group simultaneously so that the members are treated as, and behave as, peers in executing the methods invoked on the object group. Therefore, for the **ACTIVE** and **ACTIVE_WITH_VOTING** Replication Styles, the Fault Tolerance Infrastructure at the server can create an object group reference that contains profiles for gateways that multicast the request to all of the members of the object group.

27.2.5 *Accessing Server Object Groups*

The interoperable object group references permit alternative implementation strategies for connecting a client to a server object group. This section illustrates some of these strategies:

- Access via IIOP directly to a member of a server object group.
- Access via IIOP and a gateway.
- Access via a proprietary multicast group communication protocol.

27.2.5.1 *Access via IIOP Directly to the Primary Member*

This strategy may be used to provide access to a fault-tolerant server (server object group) by an unreplicated client or by a client supported by a Fault Tolerance Infrastructure from a vendor different from the vendor that provided the Fault Tolerance Infrastructure for the server. Because the access is directly to the primary member, this strategy may be used only if the server object group has the **STATELESS**, **COLD_PASSIVE**, or **WARM_PASSIVE** Replication Style.

The client ORB extracts an IIOP profile from the object group reference, preferably the profile containing the **TAG_FT_PRIMARY** component, and establishes a connection to the endpoint addressed by that profile. If the addressed endpoint is the primary member of the object group, it accepts the connection and processes the request. Otherwise, it replies with a **LOCATION_FORWARD_PERM** that provides the current object group reference, one profile of which (the one with the **TAG_FT_PRIMARY** component) contains a profile that addresses the current primary.

27.2.5.2 Access via IIOP and a Gateway

This strategy may be used to provide access to a fault-tolerant server (server object group) by an unreplicated client hosted by a non-fault-tolerant ORB and also by a client supported by a Fault Tolerance Infrastructure from a vendor different from the vendor that provided the Fault Tolerance Infrastructure for the server.

The client ORB extracts an IIOP profile from the object group reference and uses that reference to establish a connection to the endpoint addressed by that profile. If that endpoint is a gateway, it accepts the connection and forwards messages to the members of the object group, typically using a (proprietary) multicast group communication protocol.

The client ORB and the client application object must be unaware of whether the interoperable object group reference addressed a gateway or the primary member.

27.2.5.3 Access via a Multicast Group Communication Protocol

Some vendors may choose to use a proprietary multicast group communication protocol within a fault tolerance domain, or even between fault tolerance domains supported by a Fault Tolerance Infrastructure from the same vendor.

The fault tolerance domain identifier and object group identifier contained in the **TAG_FT_GROUP** component of the profiles of the object group reference could be used to establish a connection using the proprietary multicast group communication protocol. The details of connection establishment, and recovery from faults during connection establishment, for the multicast group communication protocol are not defined in this specification.

The use of a proprietary multicast group communication protocol must, however, be invisible to both the client application object and the server application object.

27.2.6 Extensions to CORBA Failover Semantics

The failover semantics for Fault Tolerant CORBA extend the failover semantics for the CORBA core, and are summarized in Table 27-1 on page 27-21. Note that the Fault Tolerant CORBA failover semantics permit reinvocation of requests even when a prior invocation yielded COMPLETED_MAYBE, whereas the CORBA failover semantics permit reinvocation only if all prior attempts yielded COMPLETED_NO. The permissible failover behaviors are determined by whether the IOR contains the **TAG_FT_GROUP** component (defined in Section 27.2.2.1, "TAG_FT_GROUP Component," on page 27-14) and whether the client ORB includes an **FT_REQUEST** service context (defined in Section 27.2.8.1, "FT_REQUEST Service Context," on page 27-24) in its request, as well as by the completion status returned and by the exception raised.

If a client tries to establish a connection to an endpoint that cannot handle the request, the client ORB might receive a reply containing a **LOCATION_FORWARD_PERM** response, which provides the most recent object group reference for the group (as described in Section 27.2.7, “Most Recent Object Group Reference,” on page 27-22), or it might receive a SYSTEM_EXCEPTION.

Each time a client ORB attempts to establish a connection, it must not abandon the attempt and raise an exception to the client application until it has tried to invoke the server using all of the alternative IIOP addresses in the IOR, and has failed to establish a connection within the **request_duration_policy_value** (defined in Section 27.2.8.2, “Request Duration Policy,” on page 27-25). It must then return a SYSTEM_EXCEPTION to the client application. Alternative addresses include all of the host/port pairs in all of the **TAG_INTERNET_IOP** profiles within the interoperable object group reference, and all of the **TAG_ALTERNATE_IIOP_ADDRESS** components.

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 IIOP addresses in the interoperable object group reference, or has received a “non-failover” condition, or the request duration has expired.

No order is prescribed for the use of the addresses present in an interoperable object group reference (including the **TAG_ALTERNATE_IIOP_ADDRESS**). If a failover condition arises, an ORB may retry with the same address, or may immediately retry with other addresses - this is a quality of implementation issue.

This behavior specifies the minimum failover semantics that an ORB must implement. An ORB may also retry in other conditions not stated above, but this is not mandated. Under all failover conditions, at most once semantics must be guaranteed.

Table 27-1 Permitted Failover Conditions without and with Transparent Reinvocation.

	Completion Status	**CORBA Exception**
Without Transparent Reinvocation	COMPLETED_NO	COMM_FAILURE TRANSIENT NO_RESPONSE OBJ_ADAPTER
With Transparent Reinvocation	COMPLETED_NO COMPLETED_MAYBE	COMM_FAILURE TRANSIENT NO_RESPONSE OBJ_ADAPTER

27.2.7 Most Recent Object Group Reference

This section defines a mechanism that allows the server to determine whether the client is using the most recent object group reference for the server object group when the client issues a request. The mechanism consists of an **FT_GROUP_VERSION** service context that a client may include in its request messages.

27.2.7.1 FT_GROUP_VERSION Service Context

The **FT_GROUP_VERSION** service context contains the version number of the object group reference for the server object group, which allows the server to determine whether the client is using an obsolete object group reference.

```
module IOP {
    const ServiceId FT_GROUP_VERSION = 12;
};

module FT {
    struct FTGroupVersionServiceContext { //context_id = FT_GROUP_VERSION;
            ObjectGroupRefVersion object_group_ref_version;
    };
};
```

If the server determines that the client is using an obsolete object group reference, the server returns a LOCATION_FORWARD_PERM response that contains the most recent object group reference for the server object group.

const ServiceId FT_GROUP_VERSION = 12;

A constant that designates the **FT_GROUP_VERSION** service context.

```
struct FTGroupVersionServiceContext{ //context_id = FT_GROUP_VERSION;
     ObjectGroupRefVersion object_group_ref_version;
};
```

A structure that contains the same **object_group_ref_version** 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, which allows the server ORB to determine whether the object group reference being used by the client is obsolete.

When the Replication Manager generates a new object group reference for the server object group, because the membership of the server object group has changed, it updates the **object_group_ref_version** in the reference for the new membership.

If the highest **object_group_ref_version** known to the server ORB is greater than that contained in the request from the client, the server ORB must return a **LOCATION_FORWARD_PERM** response to the client containing the most recent reference for the server object group.

If the **object_group_ref_version** known to the server ORB is equal to that contained in the request from the client and the server ORB supports the primary member of the server object group, the server ORB invokes the member to process the request. If the **object_group_ref_version** known to the server ORB is equal to that contained in the request from the client and the server ORB supports a backup member, the server ORB returns a TRANSIENT exception with completion status COMPLETION_NO to the client ORB. The client ORB can then reinvoke the request using another profile from the object group reference.

If the most recent **object_group_ref_version** known to the server ORB is less than that contained in the request from the client, the server ORB must obtain the current reference for the server object group. If the **object_group_ref_version** in the object group reference returned by the Replication Manager is greater than that contained in the request from the client, the server ORB must return a **LOCATION_FORWARD_PERM** response to the client containing the most recent reference for the server object group. If the **object_group_ref_version** in the object group reference returned by the Replication Manager is less than that contained in the request from the client, the server ORB returns an INV_OBJREF exception to the client.

27.2.8 Transparent Reinvocation

This section defines mechanisms that provide transparent reinvocation of methods contained in request messages. The mechanisms handle failure of the primary member of a server object group that has the **COLD_PASSIVE** or **WARM_PASSIVE** Replication Styles and provide redirection of the client's outstanding request to a backup server. In the absence of such mechanisms, the failure of the primary server could cause a client's request to be executed two (or more) times, once by the original primary and once by a backup that became the new primary, without the client or the server being aware of the repetition, possibly producing erroneous results.

These specifications do not change the current at-most-once invocation semantics of the CORBA object model. At the level of the application, a client makes a request once only and that request is executed at most once. At the transport level, however, a fault-tolerant client ORB can transparently retransmit a request message to a fault-tolerant server, to mask faults including both object and link faults, thus providing higher reliability. Transparent reinvocation is permitted only under the completion status and system exception conditions listed in Table 27-1 on page 27-21, and provided that both the IOP profile used for the existing request and the IOP profile used for the reinvocation contain a **TAG_FT_GROUP** component. Both the existing request message and the reinvocation request message must contain an **FT_REQUEST** service context. Neither the client application nor the server application is aware of such retransmissions. The server application executes the request at most once with no special application programming to handle repeated requests, and the client application receives its reply with no special application programming to handle exceptions. (For replicated clients communicating with replicated servers, use of a multicast group communication protocol may be appropriate because such a protocol provides stronger acknowledgment and retransmission mechanisms.)

The mechanisms defined here consist of the **FT_REQUEST** service context, which a client may include in its request messages, and the Request Duration Policy.

27.2.8.1 FT_REQUEST Service Context

The **FT_REQUEST** service context is used to ensure that a request is not executed more than once under fault conditions.

```
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 ORB to recognize that the request is a repetition of a previous request. If the request is a repetition of a previous request that the server has already executed, the server (which may be a new primary) does not reexecute the request but rather returns the reply that was generated by the prior execution (possibly by a previous primary that failed). The **expiration_time** serves as a garbage collection mechanism. It provides a lower bound on the time until which the server must honor the request and, therefore, retain the request and corresponding reply (if any) in its log.

```
const ServiceId FT_REQUEST = 13;
```

A constant that designates the FT_REQUEST service context.

```
struct FTRequestServiceContext { // context_id = FT_REQUEST;
            string                  client_id;
            long                    retention_id;
            TimeBase::TimeT         expiration_time;
};
```

A structure that contains the client identifier, retention identifier, and the expiration time of the request. Each repetition of a request must carry the same **client_id**, **retention_id**, and **expiration_time** as the original request. These fields are defined as follows:

The **client_id** uniquely identifies the client, so that repeated requests from the same client can be recognized. No mechanisms are defined for generating this unique identifier.

The **retention_id** uniquely identifies the request within the scope of the client and the **expiration_time**. The client ORB can reuse the **retention_id** provided that it guarantees uniqueness.

The **expiration_time** defines a lower bound on the time when the request will expire. Typically, the **expiration_time** is obtained by adding the **request_duration_policy_value** defined by the Request Duration Policy, to the local clock value of the client ORB.

If a server is unable to support the **expiration_time**, it may throw an INVALID_POLICY exception. Otherwise, the server must retain each request and its reply until the time (at the server) defined by the **expiration_time**. Until that time, the server must recognize requests that are repetitions of requests that have already been executed, and must return the reply to the original request rather than reinvoking the method. After that time, the server must return either the reply to the original request or a BAD_CONTEXT exception, but all replicas of the server must make the same decision about which reply to return so that the client receives only one reply.

The client ORB that has issued the request may reissue the request to the same or a different member of the server object group, but must use the **FT_REQUEST** service context with the same **retention_id** and same **expiration_time** as it used in its original request.

Before the server returns the reply for a request to the client, the Fault Tolerance Infrastructure must log the request and the reply. A backup that has become the new primary must not reply to the client until its state has been updated to include replies generated by other members of the object group, using the messages in the log.

Both the establishment of connections and the retention of requests are bounded by the expiration_time, or the client ORB’s current clock value plus the **request_duration_policy_value** if no **expiration_time** has been established. If a current connection fails, a new connection may be needed so that the request can be retransmitted to an alternative member of the server object group. The establishment of the new connection must be bounded by the **expiration_time** determined for the prior request.

27.2.8.2 Request Duration Policy

The Request Duration Policy 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 PolicyType REQUEST_DURATION_POLICY = 47;

    interface RequestDurationPolicy : 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 ORB to detect repeated requests.

The policy is defined by:

```
const PolicyType REQUEST_DURATION_POLICY = 47;
```

A constant that designates the **REQUEST_DURATION_POLICY**.

```
interface RequestDurationPolicy : Policy {
    readonly attribute TimeBase::TimeT   request_duration_policy_value;
};
```

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.

27.2.8.3 Fault Handling for GIOP Messages

The standard semantics of GIOP messages include definitions of fault conditions for messages of different types, and provisions for handling of faults by the ORBs. Fault Tolerant CORBA does not modify those semantics in normal (fault-free) conditions. For some types of GIOP messages, an ORB may attempt to retransmit the message or transmit the message to alternative destinations or over alternative transports. Such attempts are invisible to the client and server application and are bounded in time by the **request_duration_policy_value** defined for the client by the Request Duration Policy. We discuss below those GIOP messages for which fault handling is modified.

LocateRequest

If a client ORB loses an IIOP connection with a server while issuing a **LocateRequest**, or before receiving a corresponding **LocateReply**, or if it does not receive a **LocateReply** in a timely manner, then the client ORB may attempt to retransmit the message or to transmit the message to alternative destinations or over alternative transports. If the client ORB is unable to obtain a reply within the **request_duration_policy_value** of the Request Duration Policy, the client ORB must return a COMM_FAILURE system exception to the client application. It may return a COMM_FAILURE system exception before the end of that duration.

Request

If a client ORB loses the connection with a server or incurs some other kind of transport fault, the ORB may attempt to retransmit the request message, or retransmit the request message to an alternative destination or using an alternative transport, up to the expiration_time.

If a client invokes a fault-tolerant server (as indicated by the presence of the **TAG_FT_GROUP** component in the **TAG_INTERNET_IOP** profiles of the server's object group reference), the client ORB may retransmit a request if it would have otherwise returned a COMM_FAILURE, TRANSIENT, NO_RESPONSE, or OBJ_ADAPTER exception with a COMPLETED_NO or COMPLETED_MAYBE completion status to the client application. The client is protected against repeated execution by the inclusion of an **FT_REQUEST** service context in the request message, as described in Section 27.2.8.1, "FT_REQUEST Service Context," on page 27-24.

If a client invokes a non-fault-tolerant server (as indicated by the absence of a **TAG_FT_GROUP** component in the **TAG_INTERNET_IOP** profiles of its reference), the client ORB may retransmit the request only if it would have otherwise returned a COMM_FAILURE, TRANSIENT, NO_RESPONSE, or OBJ_ADAPTER exception with a COMPLETED_NO completion status to the client application.

LocateReply and Reply

Retransmission of a **LocateReply** or **Reply** message may occur either because the server ORB has not received a transport-level acknowledgment for a previous transmission or because the server ORB has received a repetition of a previous **LocateRequest** or **Request** message.

Fragment

Fragmented Request and Reply messages are handled like unfragmented Request and Reply messages.

27.2.9 Transport Heartbeats

With IIOP (TCP/IP), a problem can arise when a client invokes a method on a server, the host on which the server resides fails or the link fails, and the client ORB does not detect the TCP/IP problem and receives no reply. Typically, this problem is solved by using round-trip timeouts in the client application. Setting a timeout at the application level for each request is laborious, even if one knew approximately how long a particular method will take. An alternative solution proposed here is to send another request message on the same connection that takes a known (short) time to execute (i.e., a kind of no op).

This section therefore defines a new **TAG_FT_HEARTBEAT_ENABLED** component of the **TAG_INTERNET_IOP** profile, and adds two new policies: **Heartbeat** and **HeartbeatEnabled**.

27.2.9.1 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;
    };
};
```

The **TAG_FT_HEARTBEAT_ENABLED** component contains only a boolean.

```
const ComponentId TAG_FT_HEARTBEAT_ENABLED = 29;
```

A constant that designates the **TAG_FT_HEARTBEAT_ENABLED** component that is contained in a **TAG_INTERNET_IOP** profile.

```
struct TagFTHeartbeatEnabledTaggedComponent {
            // tag =TAG_FT_HEARTBEAT_ENABLED
      boolean heartbeat_enabled;
};
```

The **TAG_FT_HEARTBEAT_ENABLED** component may be included in a **TAG_INTERNET_IOP** profile to indicate that the endpoint is **heartbeat_enabled**.

27.2.9.2 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 PolicyType HEARTBEAT_POLICY = 48;

     struct HeartbeatPolicyValue {
               boolean                        heartbeat;
              TimeBase::TimeT                 heartbeat_interval;
              TimeBase::TimeT                 heartbeat_timeout;
     };

     interface HeartbeatPolicy : 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. The **FT_HB()** method name is reserved in CORBA for this purpose, and IDL compilers use the standard escape techniques if IDL specifications contain operations with this name.

If the corresponding reply message does not arrive at the client ORB within the **heartbeat_timeout** of the Heartbeat Policy of a client connected to a server over that connection, the ORB closes the connection for that client. The connection may remain open for other clients whose Heartbeat Policy define a larger value for the **heartbeat_timeout**.

The policy is defined by:

```
const PolicyType HEARTBEAT_POLICY = 48;
```

A constant that designates the Heartbeat Policy for the client.

```
struct HeartbeatPolicyValue {
    boolean                 heartbeat;
    TimeBase::TimeT         heartbeat_interval;
    TimeBase::TimeT         heartbeat_timeout;
};
```

The **HeartbeatPolicyValue** consists of a boolean that indicates whether the client ORB supports heartbeating, a **heartbeat_interval** that determines the frequency with which the client ORB pings the server, and a **heartbeat_timeout** that indicates the time by which the client ORB must receive a reply from the server before it closes the connection. Both the **heartbeat_interval** and the **heartbeat_timeout** use the standard **TimeBase::TimeT** representation, which uses a unit of 100 nanoseconds.

```
interface HeartbeatPolicy : Policy {
    readonly attribute HeartbeatPolicyValue heartbeat_policy_value;
};
```

A server ORB must respond to requests that contain the **FT_HB()** method by immediately sending a reply message. The contents of the reply message are not defined. The request id of the reply message must match the **request_id** of the request message.

A server ORB must not involve POAs or servants on receipt or reply of the **FT_HB()** message.

27.2.9.3 Heartbeat Enabled Policy

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

```
module FT {
    const 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**.

The policy is defined by:

```
const PolicyType HEARTBEAT_ENABLED_POLICY = 49;
```

A constant that designates the Heartbeat Enabled Policy for the server.

```
interface HeartbeatEnabledPolicy : Policy {
        readonly attribute boolean heartbeat_enabled_policy_value;
};
```

The **heartbeat_enabled_policy_value** determines whether the server endpoint supports heartbeats.

If a client attempts to apply the Heartbeat Policy to a server for which the Heartbeat Enabled Policy is not enabled (i.e., **heartbeat_enabled_policy_value** is false), then an INVALID_POLICIES exception is thrown. The Heartbeat Enabled Policy can be checked using **validate_policies()**.

27.3 Replication Management

27.3.1 Overview

The Replication Manager is an important component of the Fault Tolerance Infrastructure that interacts with other components of the infrastructure. Typically, the Replication Manager is replicated for fault tolerance, though not necessarily on every host within the fault tolerance domain; however, logically, there is a single Replication Manager for each fault tolerance domain.

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

The **PropertyManager** interface allows properties of the object groups to be set, such as the **ReplicationStyle**, **MembershipStyle**, **ConsistencyStyle**, **InitialNumberReplicas**, **MinimumNumberReplicas**, etc. These properties may be set statically as defaults for the fault tolerance domain or for a particular type, or may be set or changed dynamically while the application is executing.

The **GenericFactory** interface is used by the application to create object groups, as shown in Figure 27-4 on page 27-31. It is also used by the Replication Manager to create individual members of an object group.

For the infrastructure-controlled Membership Style, the Replication Manager invokes the individual factories, for the appropriate locations, to create the members of the object group, both initially to satisfy the **InitialNumberReplicas** property, and also after the loss of a member because of a fault to satisfy the **MinimumNumberReplicas** property. The Replication Manager adds the members to the object group and creates the object group reference. Subsequently, the Replication Manager removes members, if necessary.

For the application-controlled Membership Style, the **ObjectGroupManager** interface allows the application to create a member of an object group, to add an existing object to an object group, or to remove a member from an object group, citing the location of the member to be created, added or removed. It also allows the application to define the primary member of an object group and to query the locations of the members of an object group and the primary member.

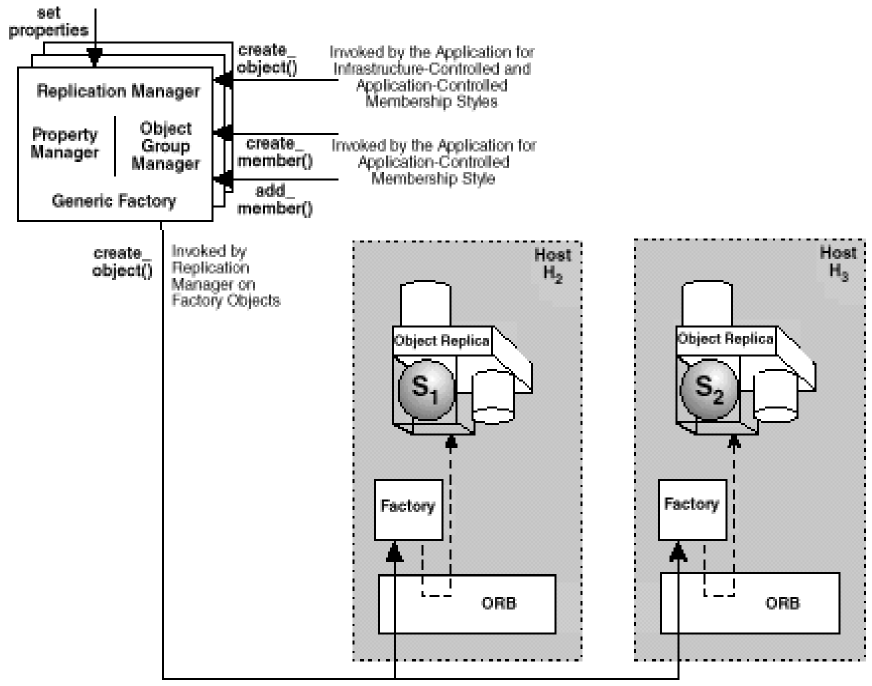
set properties
Replication Manager
Property Manager
Object Group Manager
Generic Factory
create_ object()
Invoked by the Application for Infrastructure-Controlled and Application-Controlled Membership Styles
create_ member()
add_ member()
Invoked by the Application for Application-Controlled Membership Style
create_ object()
Invoked by Replication Manager on Factory Objects
Host H2
Host H3
Object Replica
S1
S2
Factory
ORB

Figure 27-4 The Replication Manager and the Creation of an Object Group.

27.3.2 Fault Tolerance Properties

Each object group has an associated set of properties that are set as defaults for the fault tolerance domain, that are set for the type of the object, that are set when the object group is created, or that are set subsequently while the application executes. The names and values of the specified properties are given below. Vendor implementations may define additional properties and may extend the property values.

27.3.2.1 ReplicationStyle

Name	org.omg.ft.ReplicationStyle
Value	FT::STATELESS FT::COLD_PASSIVE FT::WARM_PASSIVE FT::ACTIVE FT::ACTIVE_WITH_VOTING

For the STATELESS ReplicationStyle, the behavior of the object group is unaffected by its history of invocations. A typical example is a server that provides read-only access to a database.

For the COLD_PASSIVE or WARM_PASSIVE ReplicationStyles, only a single member, the primary member, executes the methods that have been invoked on the object group. The object group contains additional backup members for recovery after a fault.

For the COLD_PASSIVE ReplicationStyle, the state of the primary is extracted from a log and loaded into a backup member when needed for recovery.

For the WARM_PASSIVE ReplicationStyle, the state of the primary member is loaded into one or more backup members periodically during normal operation.

For the ACTIVE ReplicationStyle, all of the members of the object group independently execute the methods invoked on the object, so that if a fault prevents one member from operating correctly, the other members will produce the required replies without the delay required for recovery. Duplicate requests and duplicate replies, generated by multiple members of the object group, are detected and suppressed. The ACTIVE ReplicationStyle typically requires the use of a multicast group communication system that provides reliable totally-ordered message delivery and group membership services in a model of virtual synchrony (see the Glossary).

For a source object group that has the ACTIVE_WITH_VOTING ReplicationStyle, the requests (replies) from the members of the source object group are voted, and are delivered to the members of the destination object group only if a majority of the requests (replies) are identical (match exactly). A vote on a specific request or reply must be performed using the same voting membership at each host where that vote is performed. This ReplicationStyle requires protection against commission faults both in the objects and in the network infrastructure. The ACTIVE_WITH_VOTING ReplicationStyle is not supported in the current specification, but is an anticipated extension. It should be understood that voting itself is computationally inexpensive but that the communication required to support voting properly is substantially more expensive than that required to tolerate only crash faults.

27.3.2.2 MembershipStyle

Name	org.omg.ft.MembershipStyle
Value	FT::MEMB_APP_CTRL FT::MEMB_INF_CTRL

If the value of the MembershipStyle is MEMB_APP_CTRL, the application may create an object itself and then invoke the **add_member()** method of the **ObjectGroupManager** interface to cause the Replication Manager to add the object to the object group. Alternatively, the application may invoke the **create_member()** method of the **ObjectGroupManager** interface to cause the Replication Manager to create the member and add it to the object group. The application is responsible for enforcing the **InitialNumberReplicas** and **MinimumNumberReplicas** properties. The Replication Manager initiates monitoring of the members for faults, according to the **FaultMonitoringStyle**, and registers with the Fault Notifier to receive notifications of faults. Likewise, the application may register for fault notifications for the members of the object group.

At most one member of an object group can exist at a given location. Therefore, if the application attempts to create or add a second member to an object group at the given location, a MemberAlreadyPresent exception is raised.

If the value of the **MembershipStyle** is MEMB_INF_CTRL, the Replication Manager invokes the individual factories, for the appropriate locations, to create the members of the object group, both initially to satisfy the **InitialNumberReplicas** property, and also after the loss of a member because of a fault to satisfy the **MinimumNumberReplicas** property. The Replication Manager initiates monitoring of the members for faults, according to the **FaultMonitoringStyle**, and registers with the Fault Notifier to receive notifications of faults.

27.3.2.3 ConsistencyStyle

Name	org.omg.ft.ConsistencyStyle
Value	FT::CONS_APP_CTRL FT::CONS_INF_CTRL

If the value of the **ConsistencyStyle** is CONS_APP_CTRL, the application is responsible for checkpointing, logging, activation and recovery, and for maintaining whatever kind of consistency is appropriate for the application.

If the value of the **ConsistencyStyle** is CONS_INF_CTRL, the Fault Tolerance Infrastructure is responsible for checkpointing, logging, activation and recovery, and for maintaining Strong Replica Consistency, Strong Membership Consistency, and also Uniqueness of the Primary for the COLD_PASSIVE and WARM_PASSIVE ReplicationStyles. The CONS_INF_CTRL **ConsistencyStyle** requires the object to inherit the **Checkpointable** interface.

For the COLD_PASSIVE and WARM_PASSIVE ReplicationStyles, Strong Replica Consistency requires that, at the end of each state transfer, each of the members of an object group has, or has access to, the same state and the same requests the primary replica had, or had not, processed when it created that state. It requires that requests and replies are not lost in the event of a fault and that duplicate requests and duplicate replies, generated during recovery, are suppressed.

For the ACTIVE and ACTIVE_WITH_VOTING ReplicationStyles, Strong Replica Consistency requires that, at the end of each method invocation on the object group, the members of the object group have the same state, and that no requests or replies are lost or duplicated.

For the ACTIVE, COLD_PASSIVE and WARM_PASSIVE ReplicationStyles, the behavior of each member of an object group must be deterministic and each member must start in the same state. If the same sequence of requests are then applied, in the same order, to each member of the group, Strong Replica Consistency will be maintained. Strong Replica Consistency simplifies the application programming, but requires strong mechanisms within the Fault Tolerance Infrastructure to do so. In particular, the ACTIVE and ACTIVE_WITH_VOTING ReplicationStyles, and perhaps also the WARM_PASSIVE ReplicationStyle, typically employ a multicast group communication protocol that provides reliable totally-ordered delivery of messages and group membership services to maintain Strong Replica Consistency.

Strong Membership Consistency requires that, for each method invocation on an object group, the Fault Tolerance Infrastructures on all hosts have the same view of the membership of the object group. For the COLD_PASSIVE and WARM_PASSIVE ReplicationStyles, Uniqueness of the Primary requires that there is exactly one primary member of the object group at each logical point in time.

27.3.2.4 FaultMonitoringStyle

Name	org.omg.ft.FaultMonitoringStyle
Value	FT::PULL FT::PUSH FT::NOT_MONITORED

For the PULL **FaultMonitoringStyle**, the Fault Monitor interrogates the monitored object periodically to determine whether it is alive. The PULL **FaultMonitoringStyle** requires that the object inherits the **PullMonitorable** interface.

For the PUSH **FaultMonitoringStyle**, the monitored object periodically reports to the fault monitor to indicate that it is alive. The PUSH **FaultMonitoringStyle** is not supported in the current specification, but is an anticipated extension.

27.3.2.5 FaultMonitoringGranularity

Name	org.omg.ft.FaultMonitoringGranularity
Value	FT::MEMB FT::LOC FT::LOC_AND_TYPE

For the MEMB **FaultMonitoringGranularity**, each individual member of this object group is monitored. This is the default.

For the LOC **FaultMonitoringGranularity** and for a member of this object group at a particular location, if no other object at that location is already being monitored, then the member of this object group at that location is monitored. This member acts as a “fault monitoring representative” for the members of the other objects groups at that location. If another object at that location is already being monitored, then that object acts as the “fault monitoring representative” for the member of this object group at that location. If the “fault monitoring representative” at a particular location ceases to exist due to a fault, then the Replication Manager regards all objects at that location to have failed and performs recovery for all objects at that location. If the “fault monitoring representative” ceases to exist because the member was removed from the group but had not actually failed, then the Replication Manager selects another object at that location as the “fault monitoring representative.”

For the LOC_AND_TYPE **FaultMonitoringGranularity** and for a member of this object group at a particular location, if no other object of the same type at that location is already being monitored, then the member of this object group at that location is monitored. This member acts as a "fault monitoring representative" for the members of the other object groups of the same type at that location. If another object of the same type at that location is already being monitored, then that object acts as the "fault monitoring representative" for the member of this object group at that location. If the "fault monitoring representative" at a particular location for a particular type ceases to exist due to a fault, then the Replication Manager regards all objects at that location of that type to have failed and performs recovery for all objects of that type at that location. If the "fault monitoring representative" ceases to exist because the member was removed from the group but had not actually failed, then the Replication Manager selects another object at that location of that type as the "fault monitoring representative."

27.3.2.6 Factories

Name	org.omg.ft.Factories
Value	FactoryInfos

A factory is an object, the purpose of which is to create other objects. FactoryInfos is a sequence of FactoryInfo, where FactoryInfo contains the reference to the factory, the location at which the factory is to create a member of the object group and criteria that the factory is to use to create the member.

27.3.2.7 InitialNumberReplicas

Name	org.omg.ft.InitialNumberReplicas
Value	An unsigned short

The number of replicas of an object to be created initially.

27.3.2.8 MinimumNumberReplicas

Name	org.omg.ft.MinimumNumberReplicas
Value	An unsigned short

The smallest number of replicas of an object needed to maintain the desired fault tolerance.

27.3.3 FaultMonitoringIntervalAndTimeout

Name	org.omg.ft.FaultMonitoringIntervalAndTimeout
Value	TimeBase::TimeT TimeBase::TimeT

The value is a struct that contains the interval of time between successive pings of an object, and the time allowed for subsequent responses from the object to determine whether it is faulty. TimeBase::TimeT is a long long, and the value is in units of 100 nanoseconds. FaultMonitoringInterval requires that the object inherits the **PullMonitorable** interface.

27.3.4 CheckpointInterval

Name	org.omg.ft.CheckpointInterval
Value	TimeBase::TimeT

An interval of time between writing the full state of the object to the log. **TimeBase::TimeT** is a **long long**, and the value is in units of 100 nanoseconds. **CheckpointInterval** requires that the object inherits the **Checkpointable** interface.

Note that some of these properties are incompatible, such as the STATELESS **ReplicationStyle** and **CheckpointInterval** or the CONS_APP_CTRL **ConsistencyStyle** and **CheckpointInterval**.

Table 27-2 Fault Tolerance Properties and When They May Be Set.

	Default	**Type**	**Creation**	**Dynamically**
ReplicationStyle	√	√	√	
MembershipStyle	√	√	√	
ConsistencyStyle	√	√		
FaultMonitoringStyle	√	√		
FaultMonitoringGranularity	√	√	√	√
Factories		√	√	√
InitialNumberReplicas	√	√	√	
MinimumNumberReplicas	√	√	√	√
FaultMonitoringInterval	√	√	√	√
CheckpointInterval	√	√	√	√

Table 27-2 shows the Fault Tolerance Properties and when they may be set. Properties of object groups that are set as defaults apply to all object groups of all types within a fault tolerance domain. Properties of object groups that are set for a particular type apply to all object groups of that type within the fault tolerance domain, and override the properties that are set as defaults for that type. Properties of an object group that are set at creation time are set when the particular object group is created, and override the properties that are set as defaults or for the type of the object group. Properties of an object group that are set dynamically are set while the application is executing, and override the properties that are set as defaults or for the type of the object group or when the object group is created.

27.3.5 Common Types

```
module FT {
interface GenericFactory;
interface FaultNotifier;

typedef CORBA::RepositoryId TypeId;
typedef Object ObjectGroup;

typedef CosNaming::Name Name;
typedef any Value;
struct Property {
            Name nam;
            Value val;
};
typedef sequence<Property> Properties;

typedef Name Location;
typedef sequence<Location> Locations;
typedef Properties Criteria;
struct FactoryInfo {
        GenericFactory factory;
        Location the_location;
        Criteria the_criteria;
};
typedef sequence<FactoryInfo>  FactoryInfos;

typedef long ReplicationStyleValue;
const ReplicationStyleValue STATELESS = 0;
const ReplicationStyleValue COLD_PASSIVE = 1;
const ReplicationStyleValue WARM_PASSIVE = 2;
const ReplicationStyleValue  ACTIVE = 3;
const ReplicationStyleValue  ACTIVE_WITH_VOTING = 4;

typedef long MembershipStyleValue;
const MembershipStyleValue MEMB_APP_CTRL = 0;
const MembershipStyleValue MEMB_INF_CTRL = 1;

typedef long ConsistencyStyleValue;
const ConsistencyStyleValue CONS_APP_CTRL = 0;
const ConsistencyStyleValue CONS_INF_CTRL = 1;

typedef long FaultMonitoringStyleValue;
const FaultMonitoringStyleValue PULL = 0;
const FaultMonitoringStyleValue PUSH = 1;
const FaultMonitoringStyleValue NOT_MONITORED = 2;

typedef long FaultMonitoringGranularityValue;
```

```
    const FaultMonitoringGranularityValue MEMB = 0;
    const FaultMonitoringGranularityValue LOC = 1;
    const FaultMonitoringGranularityValue LOC_AND_TYPE = 2;

    typedef FactoryInfos FactoriesValue;

    typedef unsigned short InitialNumberReplicasValue;
    typedef unsigned short MinimumNumberReplicasValue;

    struct FaultMonitoringIntervalAndTimeoutValue {
            TimeBase::TimeT monitoring_interval;
            TimeBase::TimeT timeout;
    };

    typedef TimeBase::TimeT CheckpointIntervalValue;
    exception InterfaceNotFound {};
    exception ObjectGroupNotFound {};
    exception MemberNotFound {};
    exception ObjectNotFound {};
    exception MemberAlreadyPresent {};
    exception BadReplicationStyle {};
    exception ObjectNotCreated {};
    exception ObjectNotAdded {};
    exception PrimaryNotSet {};
    exception UnsupportedProperty {
                Name nam;
                Value val;
    };
    exception InvalidProperty {
                Name nam;
                Value val;
    };
    exception NoFactory {
                Location the_location;
                TypeId type_id;
    };
    exception InvalidCriteria {
                Criteria invalid_criteria;
    };
    exception CannotMeetCriteria {
                Criteria unmet_criteria;
    };
};
```

27.3.5.1 Identifiers

typedef Object ObjectGroup;

A reference to an object group.

typedef CosNaming::Name Name;

The name of a property;

typedef any Value;

The value of a property.

```
struct Property {
Name nam;
Value val;
};
```

The name-value pair for a property. The name may be hierarchical.

typedef sequence<Property> Properties;

A sequence of properties.

typedef Name Location;

The name for a fault containment region, host, device, cluster of hosts, etc., which may be hierarchical. For example, the kind field of the name might be "HostIP" which defines a particular format for the address in the id field. The id field would then contain an IP address for a host. For each object group and each location, only one member of that object group may exist at that location.

typedef sequence<Location> Locations;

A sequence of locations of the members of an object group.

typedef Properties Criteria;

Criteria is a sequence of property (i.e., name-value pair). Examples of criteria are initialization values, constraints on an object, preferred location of the object, and fault tolerance properties of an object group.

Two names are reserved for criteria: org.omg.ft.ObjectLocation and org.omg.ft.FTProperties. The **org.omg.ft.FTProperties** name tags a location value at which an object is to be created by a factory. The **org.omg.ft.FTProperties** name tags a sequence of name-value pairs that represent fault tolerance properties for an object group. All other criteria are implementation-specific and are interpreted only by the factory.

```
struct FactoryInfo {
GenericFactory factory;
Location the_location;
Criteria the_criteria;
};
```

A structure that contains the reference to a factory and the location and the criteria that the factory uses to create an object at the given location using the given criteria, such as initialization values, constraints on the object, etc.

typedef sequence<FactoryInfo> FactoryInfos;

A sequence of FactoryInfos.

```
typedef long ReplicationStyleValue;
    const ReplicationStyleValue STATELESS = 0;
    const ReplicationStyleValue COLD_PASSIVE = 1;
    const ReplicationStyleValue WARM_PASSIVE = 2;
    const ReplicationStyleValue  ACTIVE = 3;
    const ReplicationStyleValue  ACTIVE_WITH_VOTING = 4;
```

The values of the ReplicationStyle property.

```
typedef long MembershipStyleValue;
    const MembershipStyleValue MEMB_APP_CTRL = 0;
    const MembershipStyleValue MEMB_INF_CTRL = 1;
```

The values of the MembershipStyle property.

```
typedef long ConsistencyStyleValue;
    const ConsistencyStyleValue CONS_APP_CTRL = 0;
    const ConsistencyStyleValue CONS_INF_CTRL = 1;
```

The values of the ConsistencyStyle property.

```
typedef long FaultMonitoringStyleValue;
    const FaultMonitoringStyleValue PULL = 0;
    const FaultMonitoringStyleValue PUSH = 1;
    const FaultMonitoringStyleValue NOT_MONITORED = 2;
```

The values of the FaultMonitoringStyle property.

```
typedef long FaultMonitoringGranularityValue;
    const FaultMonitoringGranularityValue MEMB = 0;
    const FaultMonitoringGranularityValue LOC = 1;
    const FaultMonitoringGranularityValue LOC_AND_TYPE = 2;
```

The values of the **FaultMonitoringGranularity** property.

typedef FactoryInfos FactoriesValue;

The value of the **Factories** property.

typedef unsigned short InitialNumberReplicasValue;

The value of the **InitialNumberReplicas** property.

typedef unsigned short MinimumNumberReplicasValue;

The value of the **MinimumNumberReplicas** property.

```
struct FaultMonitoringIntervalAndTimeoutValue {
    TimeBase::TimeT monitoring_interval;
    TimeBase::TimeT timeout;
```

};

The value of the **FaultMonitoringIntervalAndTimeout** property. Each field is of type **TimeBase::TimeT**, which is a **long long**, and is in units of 100 nanoseconds.

typedef TimeBase::TimeT CheckpointIntervalValue;

The value of the **CheckpointInterval** property. **TimeBase::TimeT** is a **long long**, and the value is in units of 100 nanoseconds

27.3.5.2 Exceptions

exception InterfaceNotFound {};

The object with the given interface is not found by the Replication Manager.

exception ObjectGroupNotFound {};

The object group with the given identifier is not found by the Replication Manager.

exception MemberNotFound {};

No member of the object group exists at the given location.

exception ObjectNotFound {};

The object is not found by the Replication Manager.

exception MemberAlreadyPresent {};

A member of the object group already exists at the given location.

exception BadReplicationStyle {};

The ReplicationStyle of the object group is inappropriate.

exception ObjectNotCreated {};

The GenericFactory did not create the object.

exception ObjectNotAdded {};

The Replication Manager did not add the object to the object group.

exception PrimaryNotSet {};

The Replication Manager did not set the primary member of the object group.

```
exception UnsupportedProperty {
      Name nam;
      Value val;
};
```

A property named in the property sequence is not supported.

```
exception InvalidProperty {
      Name nam;
      Value val;
};
```

A property value in the property sequence is not valid either in itself (for example, because the number of replicas is negative) or because it conflicts with another property in the sequence or with other properties already in effect that are not overridden.

```
exception NoFactory {
      Location the_location;
      TypeId type_id;
};
```

The factory cannot create an object at the given location with the given repository identifer.

```
exception InvalidCriteria {
      Criteria invalid_criteria;
};
```

The factory does not understand the given criteria.

```
exception CannotMeetCriteria {
      Criteria unmet_criteria;
};
```

The factory understands the given criteria, but cannot satisfy the criteria.

27.3.6 Replication Manager

The Replication Manager inherits three application program interfaces: **PropertyManager**, **ObjectGroupManager,** and **GenericFactory**. The methods inherited from the **PropertyManager** interface allow definition of properties associated with object groups created by the Replication Manager. The methods inherited from the **ObjectGroupManager** interface allow an application to exercise control over the addition, removal and location of members of an object group. The methods inherited from the **GenericFactory** interface allow the Replication Manager to create and delete object groups.

The **Replication Manager** interface provides methods that allow the Fault Notifier to register with the Replication Manager and that allow the application or Fault Tolerance Infrastructure to get the reference of the Fault Notifier subsequently. This interface may be extended with similar methods for other components of the Fault Tolerance Infrastructure by the vendors of the Fault Tolerance Infrastructure.

Note that the **ReplicationManager** interface does not contain **register_fault_monitor()** or **get_fault_monitor()** methods. The reason is that typically there will be several fault monitors (detectors) within a fault tolerance domain, for example, a fault detector on each of the individual hosts that monitors the objects on that host, and a fault detector for the fault tolerance domain that monitors the fault detectors and the hosts within that domain. Therefore, the means of obtaining the references to the fault monitors is not specified. The Naming Service or Trader Service could be used to obtain the references to the various fault monitors.

```
module FT {
    interface ReplicationManager : PropertyManager, ObjectGroupManager,
            GenericFactory {
        void register_fault_notifier(in FaultNotifier fault_notifier);

     FaultNotifier get_fault_notifier()
         raises (InterfaceNotFound);
    };
};
```

27.3.6.1 Operations

register_fault_notifier

This method registers the Fault Notifier with the Replication Manager.

```
void register_fault_notifier(in FaultNotifier fault_notifier);
```

Parameters

fault_notifier	The reference of the Fault Notifier that is to be registered.

get_fault_notifier

This method returns the reference of the Fault Notifier.

```
FaultNotifier get_fault_notifier()
    raises (InterfaceNotFound);
```

Return Value

The reference of the Fault Notifier.

Raises

InterfaceNotFound if the Fault Notifier is not found.

27.3.7 PropertyManager

The **PropertyManager** interface provides methods that set properties for object groups, such as the **ReplicationStyle**, **MembershipStyle**, **ConsistencyStyle**, **InitialNumberReplicas**, **MinimumNumberReplicas**, etc. It may set these properties statically as defaults for the fault tolerance domain or for a particular type, or may set or change the properties dynamically while the application is executing.

```
module FT {
    interface PropertyManager {
                 void set_default_properties(in Properties props)
                      raises (InvalidProperty,
                          UnsupportedProperty);

        Properties get_default_properties();

        void remove_default_properties(in Properties props)
            raises (InvalidProperty,
                UnsupportedProperty);

        void set_type_properties(in TypeId type_id,
                                   in Properties overrides)
            raises (InvalidProperty,
                UnsupportedProperty);

        Properties get_type_properties(in TypeId type_id);

        void remove_type_properties(in TypeId type_id,
                                   in Properties props)
            raises (InvalidProperty,
                UnsupportedProperty);

        void set_properties_dynamically(in ObjectGroup object_group,
                                         in Properties overrides)
            raises(ObjectGroupNotFound,
                    InvalidProperty,
                    UnsupportedProperty);

        Properties get_properties(in ObjectGroup object_group)
              raises(ObjectGroupNotFound);
        };
    };
```

27.3.7.1 Operations

set_default_properties

This method sets the default properties for all object groups that are to be created within the fault tolerance domain.

```
void set_default_properties(in Properties props)
     raises (InvalidProperty,
            UnsupportedProperty);
```

Parameters

props	The properties to be set for all newly created object groups within the fault tolerance domain.

Raises

InvalidProperty if one or more of the properties in the sequence is not valid.

UnsupportedProperty if one or more of the properties in the sequence is not supported.

get_default_properties

This method returns the default properties for the object groups within the fault tolerance domain.

```
Properties get_default_properties();
```

Return Value

The default properties that have been set for the object groups.

remove_default_properties

This method removes the given default properties.

```
void remove_default_properties(in Properties props)
        raises (InvalidProperty,
                   UnsupportedProperty);
```

Parameters

props	The properties to be removed.

Raises

InvalidProperty if one or more of the properties in the sequence is not valid.

UnsupportedProperty if one or more of the properties in the sequence is not supported.

set_type_properties

This method sets the properties that override the default properties of the object groups, with the given type identifier, that are created in the future.

```
void set_type_properties(in TypeId type_id,
                   in Properties overrides)
      raises (InvalidProperty,
                 UnsupportedProperty);
```

Parameters

type_id	The repository id for which the properties, that are to override the existing properties, are set
overrides	The overriding properties.

Raises

InvalidProperty if one or more of the properties in the sequence is not valid.

UnsupportedProperty if one or more of the properties in the sequence is not supported.

get_type_properties

This method returns the properties of the object groups, with the given type identifier, that are created in the future. These properties include the properties determined by **set_type_properties()**, as well as the default properties that are not overridden by **set_type_properties()**.

```
Properties get_type_properties(in TypeId type_id);
```

Parameters

type_id	The repository id for which the properties, that are to override the existing properties, are set

Return Value

The effective properties for the given type identifier.

remove_type_properties

This method removes the given properties, with the given type identifier.

```
void remove_type_properties(in TypeId type_id,
                            in Properties props)
        raises (InvalidProperty,
            UnsupportedProperty);
```

Parameters

type_id	The repository id for which the given properties are to be removed.
props	The properties to be removed.

Raises

InvalidProperty if one or more of the properties in the sequence is not valid.

UnsupportedProperty if one or more of the properties in the sequence is not supported.

set_properties_dynamically

This method sets the properties for the object group with the given reference dynamically while the application executes. The properties given as a parameter override the properties for the object when it was created which, in turn, override the properties for the given type which, in turn, override the default properties.

```
void set_properties_dynamically(in ObjectGroup object_group,
                                in Properties overrides)
         raises(ObjectGroupNotFound,
                InvalidProperty,
                UnsupportedProperty);
```

Parameters

object_group	The reference of the object group for which the overriding properties are set.
overrides	The overriding properties.

Raises

InvalidProperty if one or more of the properties in the sequence is invalid.

UnsupportedProperty if one or more of the properties in the sequence is not supported.

27.3.7.2 *get_properties*

This method returns the current properties of the given object group. These properties include those that are set dynamically, those that are set when the object group was created but are not overridden by **set_properties_dynamically()**, those that are set as properties of a type but are not overridden by **create_object()** and **set_properties_dyamically()**, and those that are set as defaults but are not overridden by **set_type_properties()**, **create_object()**, and **set_properties_dyamically()**.

```
Properties get_properties(in ObjectGroup object_group)
     raises(ObjectGroupNotFound);
```

Parameters

object_group	The reference of the object group for which the properties are to be returned.

Return Value

The set of current properties for the object group with the given reference.

Raises

ObjectGroupNotFound if the object group is not found by the Replication Manager.

27.3.8 ObjectGroupManager

The **ObjectGroupManager** interface provides methods that allow an application to exercise control over the addition, removal and locations of members of an object group and to obtain the current reference and identifier for an object group.

```
module FT {
    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);

        ObjectGroup set_primary_member(in ObjectGroup object_group,
                                      in Location the_location)
                raises(ObjectGroupNotFound,
                        MemberNotFound,
                        PrimaryNotSet,
                        BadReplicationStyle);

        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);
        };
};
```

27.3.8.1 Operations

create_member

The **create_member()** method allows the application to exercise explicit control over the creation of a member of an object group, and to determine where the member is created.

```
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);
```

Parameters

object_group	The object group reference for the object group to which the member is to be added.
the_location	The physical location (i.e., a fault containment region, host, cluster of hosts, etc.) at which the new member is to be created. There is at most one member of an object group at each location.
type_id	The repository identifier for the type of the object.
the_criteria	Parameters to be passed to the factory, which the factory evaluates before creating the object. The criteria are implementation-specific and are not defined in this specification. Examples of criteria are initialization values, constraints on the member, etc. The criteria passed in as a parameter to **create_member()**, if any, override the criteria set in the FactoryInfos property of the given object group for the given location.

Return Value

The object group reference of the object group with the member added. This reference may be the same as that passed in as a parameter.

Raises

ObjectGroupNotFound if the object group is not found by the Replication Manager.

MemberAlreadyPresent if a member of the object group already exists at the given location.

NoFactory if the Replication Manager cannot find a factory that is capable of constructing a member of the object group with the given type_id and at the given location.

ObjectNotCreated if the factory or the Replication Manager cannot create the member and add it to the object group.

InvalidCriteria if the factory does not understand the critera

CannotMeetCriteria if the factory understands the criteria but cannot satisfy it.

add_member

The **add_member()** method allows an application to exercise explicit control over the addition of an existing object to an object group at a particular location.

```
ObjectGroup add_member(in ObjectGroup object_group,
                       in Location the_location,
                       in Object member)
          raises(ObjectGroupNotFound,
                 MemberAlreadyPresent,
                 ObjectNotAdded);
```

Parameters

object_group	The object group reference of the object group to which the existing object is to be added.
the_location	The physical location (i.e., a fault containment region, host, cluster of hosts, etc.) of the object to be added. There is at most one member of an object group at each location.
member	The reference of the object to be added.

Return Value

The object group reference for the object group with the object added. This reference may be the same as that passed in as a parameter.

Raises

ObjectGroupNotFound if the object group is not found by the Replication Manager.

MemberAlreadyPresent if a member of the object group already exists at the given location.

ObjectNotAdded if the Replication Manager cannot add the object to the object group.

remove_member

The **remove_member()** method allows an application to exercise explicit control over the removal of a member from an object group at a particular location.

If the application invoked the **create_object()** method of the **GenericFactory** interface to create the member object and used the **add_member()** method to add the object to the object group, when the application invokes **remove_member()**, the Replication Manager removes the member from the group but does not delete it. Deletion of the object is the responsibility of the application.

If the application invoked the **create_member()** method to create the member object, when the application invokes the **remove_member()** method to remove the member from the object group, the Replication Manager first removes the member from the object group and then invokes the **delete_object()** method of the **GenericFactory** interface to delete the object.

If the Replication Manager invoked the **create_object()** method of the **GenericFactory** interface to create the member object, when the application invokes the **remove_member()** method to remove the member, the Replication Manager first removes the member from the group and then invokes the **delete_object()** method of the **GenericFactory** interface to delete the object.

If the MembershipStyle is MEMB_INF_CTRL, the application invokes the **remove_member()** method and the number of members of the object group falls below the MinimumNumberReplicas, then the Replication Manager starts up a new member at another location.

```
ObjectGroup remove_member(in ObjectGroup object_group,
                                            in Location the_location)
                raises(ObjectGroupNotFound,
                           MemberNotFound};
```

Parameters

object_group	The object group reference of the object group from which the member is to be removed.
the_location	The physical location (i.e., a fault containment region, host, cluster of hosts, etc.) of the member to be removed.

Return Value

The object group reference for the object group with the member removed. This reference may be the same as that passed in as a parameter.

Raises

ObjectGroupNotFound if the object group is not found by the Replication Manager.

MemberNotFound if the Replication Manager cannot find a member of the object group at the given location.

set_primary_member

The **set_primary_member()** method allows the application to exercise explicit control over the selection of the member of the object group that is to be the primary.

```
ObjectGroup set_primary_member(in ObjectGroup object_group,
                                  in Location the_location)
            raises(ObjectGroupNotFound,
                   MemberNotFound,
                   PrimaryNotSet,
                   BadReplicationStyle)
```

Parameters

object_group	The object group reference of the object group whose primary is to be determined.
the_location	The physical location of the member that is to become the primary.

Return Value

The object group reference of the object group with the primary member at the given location. This reference may be the same as that passed in as a parameter.

Raises

ObjectGroupNotFound if the object group is not found by the Replication Manager.

MemberNotFound if the Replication Manager cannot find a member of the object group at that location.

PrimaryNotSet if the Replication Manager cannot set the primary member of the object group.

BadReplicationStyle if the ReplicationStyle of the given group is not COLD_PASSIVE or WARM_PASSIVE.

locations_of_members

The **locations_of_members()** method allows the application to determine the locations of the members of the given object group, and also the location of the primary member of the group.

```
Locations locations_of_members(in ObjectGroup object_group)
        raises(ObjectGroupNotFound);
```

Parameters

object_group	The object group reference of the object group.

Return Value

A sequence of locations at which the members of the object group currently exist. If the object group has the COLD_PASSIVE or WARM_PASSIVE **ReplicationStyle**, the first location in the sequence is the location of the primary.

Raises

ObjectGroupNotFound if the object group is not found by the Replication Manager.

get_object_group_id

The **get_object_group_id()** method takes a reference for an object group as an in parameter, and returns the identifier of the object group.

```
ObjectGroupId get_object_group_id(in ObjectGroup object_group)
        raises(ObjectGroupNotFound);
```

Parameters

object_group	An object group reference for the object group.

Return Value

The object group identifier for the object group.

Raises

ObjectGroupNotFound if the object group is not found by the Replication Manager.

get_object_group_ref

The **get_object_group_ref()** method takes a reference for an object group as an in parameter, and returns the current reference for the object group.

```
ObjectGroup get_object_group_ref(in ObjectGroup object_group)
```

```
raises(ObjectGroupNotFound);
```

Parameters

object_group	An object group reference for the object group.

Return Value

The current object group reference for the object group. The returned reference may be the same as the reference passed in as a parameter.

Raises

ObjectGroupNotFound if the object group is not found by the Replication Manager.

get_member_ref

The **get_member_ref()** method takes a reference for an object group and a location as in parameters, and returns a reference for the member.

```
Object get_member_ref(in ObjectGroup object_group,
                                in Location loc)
        raises(ObjectGroupNotFound,
                   MemberNotFound);
```

Parameters

object_group	An object group reference for the object group.
loc	The location of the member.

Return Value

The reference for the member.

Raises

ObjectGroupNotFound if the object group is not found by the Replication Manager.

MemberNotFound if the member is not found by the Replication Manager.

27.3.9 GenericFactory

The **GenericFactory** interface is generic in that it allows the creation of replicated objects (object groups), replicas (members of object groups), and unreplicated objects. It is inherited by the Replication Manager to allow the application to invoke the Replication Manager to create replicated objects. It is implemented by the application's local factory objects on the various hosts to allow the Replication

Manager to invoke the local factory objects of the application to create individual members of an object group and also to allow the application to invoke the local factory objects to create individual (unreplicated) objects.

The **GenericFactory** interface, inherited by the Replication Manager, is programmed by the vendor of the Fault Tolerance Infrastructure. In contrast, the local factory objects, that implement the **GenericFactory** interface, are programmed by the application programmer, rather than by the vendor of the Fault Tolerance Infrastructure; they can be regarded in the same light as the **Monitorable**, **Checkpointable**, and **Updateable** interfaces.

The **GenericFactory** interface provides **create_object()** and **delete_object()** methods for creating and deleting objects and object groups.

The application program invokes the **create_object()** method of the **GenericFactory** interface inherited by the Replication Manager to create an object group, whether it is application-controlled or infrastructure-controlled, and similarly for the **delete_object()** method.

If the MembershipStyle is MEMB_INF_CTRL, the Replication Manager in turn invokes the **create_object()** method of the **GenericFactory** interface of the appropriate local factories to create the members of the object group and then adds them to the group.

If the MembershipStyle is MEMB_APP_CTRL, the application or an application-level manager may invoke the **create_member()** method of the **ObjectGroupManager** interface which, in turn, causes the Replication Manager to invoke the **create_object()** method of the **GenericFactory** interface of the local factory, using the given location and criteria, and then to add the member to the group. Alternatively, the application or an application-level manager itself may invoke the **create_object()** method of the **GenericFactory** interface of the local factory to create the object and may then invoke the **add_member()** method of the **ObjectGroupManager** interface to cause the Replication Manager to add the object to the group.

To create an unreplicated object, the application invokes the **create_object()** method of the **GenericFactory** interface of a specific local factory.

```
module FT {
    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);
    };
};
```

There may be multiple different implementations of the **GenericFactory** interface. Each such factory implementation may create objects of one or more types at one or more locations.

The **create_object()** method takes a **type_id** as an **in** parameter. It also takes **the_criteria** as an in parameter, which allows a user to specify additional criteria, such as initialization values for the object implementation, constraints on the object, or preferred location of the object. The **type_id** and **the_criteria in** parameters of the **create_object()** method contribute to the genericity and the flexibility of the **GenericFactory** interface.

The **create_object()** method of the **GenericFactory** interface, implemented by the application's local factory objects, accepts a criterion with the reserved name org.omg.ft.ObjectLocation. The value of this criterion instructs the factory where to create the object.

The **create_object()** method of the **GenericFactory** interface, inherited by the Replication Manager, accepts fault tolerance properties within **the_criteria** parameter. These fault tolerance properties are contained in a single criterion with the reserved name org.omg.ft.FTProperties. Such properties, if any, override the corresponding fault tolerance properties that are specified as defaults or based on the type of the object. The Replication Manager removes the org.omg.ft.FTProperties criterion from **the_criteria** passed to it by the application in the **create_object()** method and adds the org.omg.ft.ObjectLocation criterion to the criteria before passing **the_criteria** as a parameter of the **create_object()** method to the application's local factory.

The **create_object()** method of the **GenericFactory** interface, implemented by the application's local factory objects, returns an object reference as a result.

The **create_object()** method of the **GenericFactory** interface, inherited by the Replication Manager, returns an object group reference as a result. If the **MembershipStyle** is MEMB_APP_CTRL, the Replication Manager creates an object group with no members. Consequently, the returned object group reference contains no **TAG_INTERNET_IOP** profiles but, instead, contains a **TAG_MULTIPLE_COMPONENTS** profile with the **TAG_FT_GROUP** component in it.

The **create_object()** method has an **out** parameter, **factory_creation_id**, that is retained by the entity that invoked the method so that it can later invoke the **delete_object()** method of the factory using the **factory_creation_id** as an **in** parameter, to cause the factory to delete the object. The factory must also retain this identification information so that it can actually delete the object.

Because the factory retains the identification information that is needed to delete an object that it created, the factory has state. The local factories that create the members of an object group are not replicas of one another. To protect each of these local factories against faults, the application deployer either may replicate each of the factories using the COLD_PASSIVE ReplicationStyle, or may assume that the failure of a local factory at a location (e.g., process or host) is equivalent to the failure of that location.

The application deployer registers a sequence of factories with the Property Manager as the Factories property of the object group, which contains a sequence of factory reference, **the_location** and **the_criteria**, which determine where the factory may create an object and the criteria for the object that it is to create.

If the MembershipStyle is MEMB_INF_CTRL, the Replication Manager uses the locations to choose one or more factories from the Factories sequence and uses the factory references to invoke the **create_object()** method of the **GenericFactory** interface that the factories implement to create the members of the object group.

If the MembershipStyle is MEMB_APP_CTRL and the application itself invokes the **create_member()** method of the **ObjectGroupManager** interface, citing a location that it selected, the Replication Manager invokes the **create_object()** method of the **GenericFactory** interface implemented by the factory (provided by the Factories property) for that location to create the new member of the object group at that location.

If the MembershipStyle is MEMB_APP_CTRL and the application invokes the **create_object()** method of the **GenericFactory** interface for a particular factory to create an object, it may then invoke the **add_member()** method of the **ObjectGroupManager** interface to add the object to the group.

Similarly, to create an unreplicated object, the application may invoke the **create_object()** method of the **GenericFactory** interface of one of its own factories.

27.3.9.1 Identifiers

typedef any FactoryCreationId;

An identifer that is assigned to an object by the factory that creates the object and that is used by the factory to delete the object subsequently.

27.3.9.2 Operations

create_object

This method of the **GenericFactory** interface creates an object, using the type_id parameter to determine which type of object to create and **the_criteria** parameter to determine restrictions on how and where to create the object. The **out** parameter, **factory_creation_id**, allows the entity that invoked the factory, and the factory itself, to identify the object for subsequent deletion.

If the application or the Replication Manager invokes the **create_object()** method on the **GenericFactory** interface, implemented by the application’s local factory object, then it creates a single object.

If the application invokes the **create_object()** method on the **GenericFactory** interface, inherited by the Replication Manager, then it creates an object group. For an object group with the MEMB_APP_CTRL MembershipStyle, the Replication Manager returns an object group reference containing only the **TAG_MULTIPLE_COMPONENTS** profile with the **TAG_FT_GROUP** component in it.

One of the name-value pairs in **the_criteria**, passed to the Replication Manager as a parameter of **create_object()**, may have the name org.omg.ft.FTProperties (which is reserved for specifying fault tolerance properties). The Replication Manager removes that entry of the sequence, adds the org.omg.ft.ObjectLocation entry (which is reserved for specifying the location at which the factory is to create the object), and appends any location-specific criteria (specified in the Factories property for the particular location) before it invokes **create_object()** method on the application's local factory object.

```
Object create_object(in TypeId type_id,
                     in Criteria the_criteria,
                     out FactoryCreationId factory_creation_id)
        raises (NoFactory,
                ObjectNotCreated,
                InvalidCriteria,
                InvalidProperty,
                CannotMeetCriteria);
```

Parameters

type_id	The repository identifier of the object to be created by the factory.
the_criteria	Information passed to the factory, which the factory evaluates before creating the object. Examples of criteria are initialization values, constraints on the object, preferred location of the object, fault tolerance properties for an object group, etc.
factory_creation_id	An identifier that allows the factory to delete the object subsequently.

Return Value

The reference to the object created by the GenericFactory. When the **GenericFactory** interface is implemented by the application's local factory object, the **create_object()** method returns an object reference as a result. When the **GenericFactory** interface is inherited by the Replication Manager, the **create_object()** method returns an object group reference as a result.

Raises

NoFactory if the object cannot be created. When the **GenericFactory** interface is implemented by the application's local factory object, the raised exception indicates that the factory cannot create an individual object of the **type_id** at the location. When the **GenericFactory** interface is inherited by the Replication Manager, the raised exception indicates that the Replication Manager cannot create the object group because it cannot find a factory that is capable of constructing a member of the object group of the **type_id** at the location.

ObjectNotCreated if the factory cannot create the object.

InvalidCriteria if the factory does not understand the criteria.

InvalidProperty if a property passed in as criteria is invalid.

CannotMeetCriteria if the factory understands the criteria but cannot satisfy it.

delete_object

This method deletes the object with the given identifier. If the application or the Replication Manager invokes this method on the **GenericFactory** interface, implemented by the application's local factory object, then it deletes a single object.

If the application invokes this method on the **GenericFactory** interface, inherited by the Replication Manager, then it deletes an object group. When this method is invoked on it, the Replication Manager must first remove each of the members from the object group, and delete each of them, before it deletes the object group itself.

```
void delete_object(in FactoryCreationId factory_creation_id)
        raises(ObjectNotFound);
```

Parameters

factory_creation_id	An identifier for the object that is to be deleted.

Raises

ObjectNotFound if the object cannot be found.

27.3.10 Obtaining the Reference for the Replication Manager

The application may obtain a reference to the Replication Manager for its Fault Tolerance Domain by invoking **resolve_initial_references()** with an ObjectId of "ReplicationManager" and narrowing to the appropriate type.

27.3.11 Use Cases

27.3.11.1 Infrastructure-Controlled Membership Style

1. The application obtains a reference to the Replication Manager by invoking **resolve_initial_references()** and narrowing the result.

2. To create a replicated object (object group), the application invokes the **create_object()** method of the **GenericFactory** interface inherited by the Replication Manager, supplying the **type_id** and **the_criteria**. The

create_object() method returns (at Step 11) the object group reference and also the object group identifier as the **factory_creation_id**, which is recorded by the application to permit it to subsequently request the **GenericFactory** to delete the object group.

3. The Replication Manager obtains the fault tolerance properties for the object group from the Property Manager of the type defined by the **type_id** parameter. If additional fault tolerance properties are defined in an entry named org.omg.ft.FTProperties of **the_criteria** parameter, those properties override the properties obtained from the Property Manager.

4. Using the **InitialNumberReplicas** property and the **Factories** property (a sequence of factory, location at which the factory is to create the object and criteria that the factory is to use in creating the object), the Replication Manager decides the locations at which to create the members of the object group.

5. For each member, the Replication Manager invokes the **create_object()** method of the **GenericFactory** interface of the requisite factory provided by the application for the location of the member, passing in as parameters the **type_id** and **the_criteria** obtained from the **Factories** property, as shown in Figure 3-2. The method returns the reference of the member and its **factory_creation_id**, which is unique within the context of the factory. The factory and the Replication Manager record this information to allow the Replication Manager to invoke the **delete_object()** method of the **GenericFactory** interface of the same local factory to delete the member subsequently.

6. The Replication Manager determines the identifier of the object group, and constructs the **TAG_FT_GROUP** component containing the fault tolerance domain identifier, the object group identifier and the object group version that allow the object group to be addressed. The Replication Manager then constructs the object group reference.

7. For each gateway:

   a. The Replication Manager constructs a **TAG_INTERNET_IOP** profile for the gateway containing its host and port, and a **TAG_FT_GROUP** component that allows the object group to be addressed.

   b. The Replication Manager then augments the object group reference with the gateway profile.

8. The Replication Manager records the object group reference for the object group against the object group identifier.

9. For each member:

   a. The Replication Manager adds the member to the object group.

   b. Depending on the Replication Style, the Replication Manager activates the member.

   c. The Replication Manager checks the Replication Style, Fault Monitoring Style, Fault Monitoring Granularity to determine whether to initiate fault monitoring of the member.

d. The Replication Manager registers itself, or a fault consumer object that it has created, with the Fault Notifier to receive notifications of faults for the member.

10. For the COLD_PASSIVE or WARM_PASSIVE Replication Styles, the Replication Manager determines the primary member of the group and includes the **TAG_FT_PRIMARY** component in the profile for that member.

11. The Replication Manager returns to the application the object group reference for the object group, as constructed in Step 7, and also the **object_group_id** as the **out** parameter, **factory_creation_id**, of the **create_object()** method.

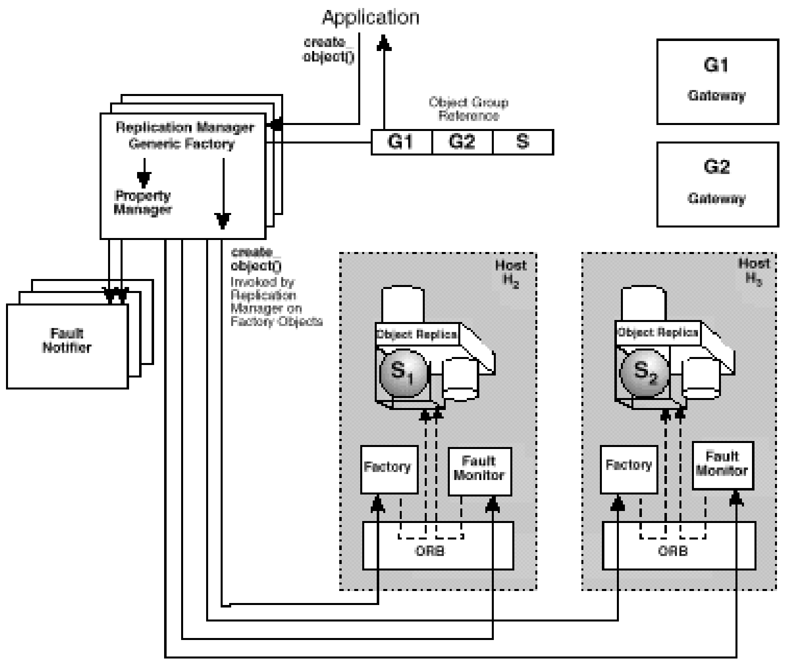

Figure 27-5 The Creation of an Object Group with the Infrastructure-Controlled MembershipStyle.

27.3.11.2 *Application-Controlled Membership Style*

1. The application obtains a reference to the Replication Manager by invoking **resolve_initial_references()**.

2. The application obtains the fault tolerance properties from the Property Manager, including the InitialNumberReplicas.

3. To create a replicated object (object group), the application invokes the **create_object()** method of the **GenericFactory** interface inherited by the Replication Manager, supplying the **type_id** and **the_criteria**, as shown in Figure 27-5.

4. The Replication Manager determines the identifier of the object group, and constructs the **TAG_FT_GROUP** component containing the fault tolerance domain identifier, the object group identifier and the object group version. The Replication Manager then constructs the object group reference, containing the **TAG_MULTIPLE_COMPONENTS** profile with the **TAG_FT_GROUP** component in it.

5. The Replication Manager returns to the application, as the reply to **create_object()**, the object group reference and the object group identifer as the **factory_creation_id**, which allows the application to delete the object group subsequently.

6. For each member:

    a. If the application has already created the object that is to become the member, the application invokes the **add_member()** method of the **ObjectGroupManager** interface, citing the object group reference, location and member reference.

    b. If instead the application wants the infrastructure to create the member, the application invokes the **create_member()** method of the **ObjectGroupManager** interface, citing the object group reference, location, **type_id** and **the_criteria**, as shown in Figure 27-6.

       The Replication Manager obtains the object reference for the **factory**, **the_location**, and **the_criteria** from the Factories property. The Replication Manager takes **the_criteria** passed to it by **create_member()**, appends the property with the name org.omg.ft.ObjectLocation and **the_location** value passed to it by **create_member()**, and appends **the_criteria** from the Factories property for the particular location. It then invokes the **create_object()** method of the **GenericFactory** interface of the factory provided by the application to create a member at that location, passing in the **type_id** and **the_criteria**.

       The factory returns the object reference and the **factory_creation_id** for the new member, and records this identification information. The Replication Manager records the **factory_creation_id**, which allows it subsequently to invoke the **delete_object()** method of the **GenericFactory** interface of the local factory to delete the member.

    c. The Replication Manager constructs a new object group reference, taking the new member into account. The new object group reference may be the same as the existing object group reference.

    d. The Replication Manager checks the **FaultMonitoringStyle**, **FaultMonitoringGranularity**, and **FaultMonitoringInterval** properties and initiates monitoring of the new member.

    e. The Replication Manager registers itself, or a fault consumer object that it has created, with the Fault Notifier to receive fault reports about the new member.

    f. The Replication Manager returns the new object group reference to the application in case (a) as the return value of **add_member()** and in case (b) as the return value of **create_member()**.

7. For the COLD_PASSIVE or WARM_PASSIVE Replication Managers, the application determines which of the members is to be the primary and invokes the **set_primary_member()** method of the **ObjectGroupManager** interface. The Replication Manager puts the **TAG_FT_PRIMARY** component in the appropriate profile of the object group reference and returns the object group reference to the application as the return value of **set_primary_member()**.

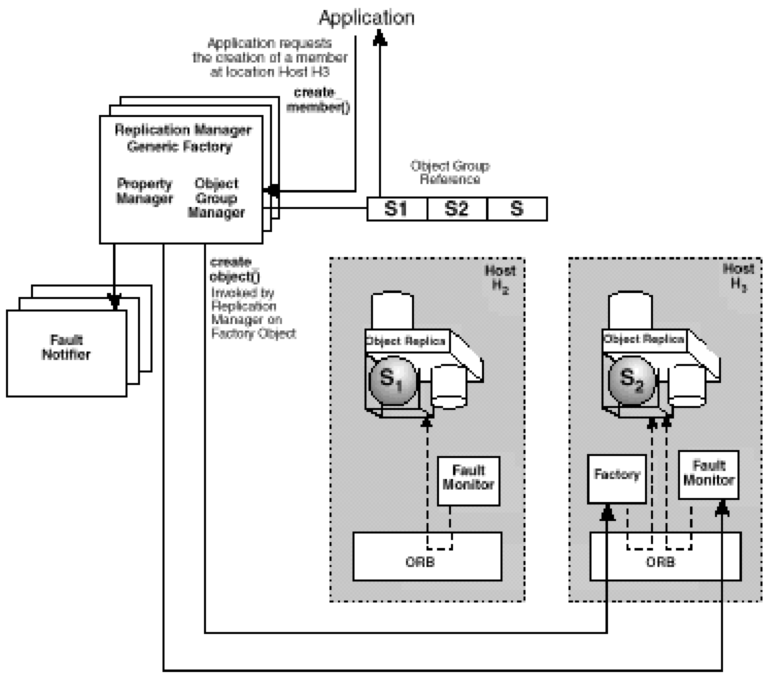

Figure 27-6 The Creation of a Member of an Object Group with the Application-Controlled Membership Style.

27.3.11.3 Unreplicated Object Creation and Deletion

Creation

1. The application obtains a reference to the local factory.
2. The application invokes the **create_object()** method of the **GenericFactory** interface of the local factory, supplying the **type_id** and **the_criteria**.
3. The factory creates the object and returns the object reference and the **factory_creation_id** to the application, as the result of **create_object()**. The **factory_creation_id** is unique within the context of the factory. The application and the factory record this identification information, which they can use subsequently to delete the object.

Deletion

1. The application invokes the **delete_object()** method of the **GenericFactory** interface of the local factory, supplying the **factory_creation_id**.
2. The factory associates the **factory_creation_id** with the recorded information and deletes the object.

27.4 Fault Management

27.4.1 Overview

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

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

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

A fault-tolerant system typically has several Fault Detectors, including those provided by the infrastructure to monitor objects, and also other fault detectors provided by the infrastructure or the application. Each Fault Detector belongs to a particular fault tolerance domain, and is not shared across fault tolerance domains. Most implementations of Fault Detectors are based on timeouts, and use either pull- or push-based monitoring. This section defines an interface for pull-based monitoring, the **PullMonitorable** interface, that application objects inherit, and that is invoked by a Fault Detector within the Fault Tolerance Infrastructure.

The section also defines a **FaultNotifier** interface. The Fault Notifier receives fault reports from the Fault Detectors. The Fault Notifier filters the reports to eliminate unnecessary or duplicate reports. It then sends fault event notifications to the consumers. The Replication Manager is such a consumer, as is the Fault Analyzer. The application can also subscribe to receive fault event notifications. Logically, there is one Fault Notifier per fault tolerance domain, although typically it is replicated for fault tolerance. The Fault Notifier belongs to a particular fault tolerance domain and is not shared across domains.

A fault-tolerant system may also have one or more Fault Analyzers. Each Fault Analyzer collects fault reports and performs event correlation, analysis, and diagnosis. It may condense a large number of related fault reports into a single fault report (e.g., the crash of a host can cause fault reports for all objects on that host, as well as a fault report for the host itself). The analysis of fault reports is application-dependent; thus, this chapter does not define a Fault Analyzer interface, but allows an application developer to hook in Fault Analyzers as consumers of fault reports generated by the Fault Notifier.

A problem with fault notification is the potential for a large number of notifications to be generated by a single fault. This problem is addressed by filtering within the Fault Notifier, by Fault Analyzers, and by the FaultMonitoringGranularity.

27.4.2 Architecture

Figure 27-7 shows the interaction between the Fault Detectors, Fault Notifier, Fault Analyzer, and Replication Manager in a relatively simple system. The fault management specification defines interfaces that allow interaction of:

- A Fault Detector with a pull-monitored object within a fault tolerance domain
- A Fault Detector with the Fault Notifier within a fault tolerance domain
- The Fault Notifier with the Replication Manager, a Fault Analyzer, or other application objects within a fault tolerance domain.

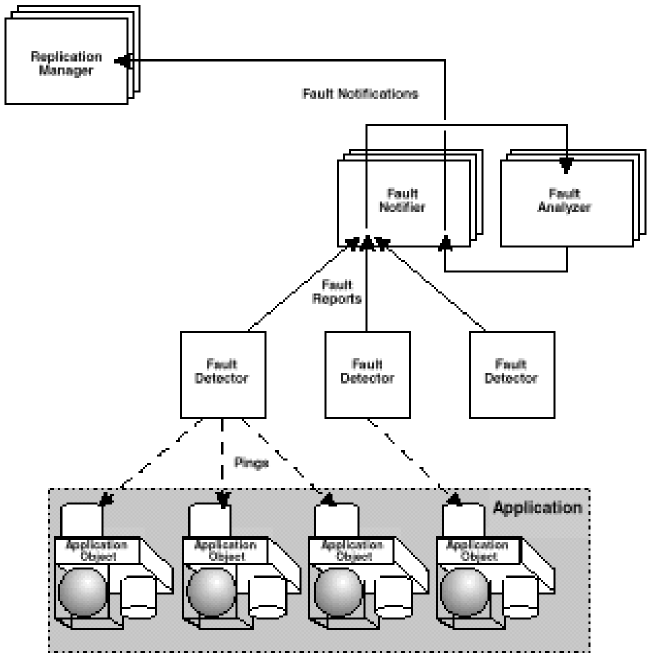

Figure 27-7 Interactions between the Fault Detectors, Fault Notifier, Fault Analyzer, and Replication Manager.

27.4.2.1 Fault Detection

In the Fault Tolerance Infrastructure, fault detection is initiated by the Replication Manager for members of object groups having either application-controlled or infrastructure-controlled MembershipStyles (see Section 27.3.2, “Fault Tolerance Properties,” on page 27-31). Because the fault management specification focuses on monitoring and timeout-based fault detection, the terms monitor and detector are used interchangeably.

There are two common styles of fault monitoring: PULL and PUSH. These two fault monitoring styles differ in the direction in which fault information flows in the system. Because push-based monitoring depends on characteristics of the application, it is not defined in this specification.

The fault management specification defines the interaction between a pull-based Fault Detector and application objects. It defines a **PullMonitorable** interface that the application objects inherit. Other kinds of system-specific (e.g., host, network) and application-specific Fault Detectors may be present in the system, but they are not defined.

27.4.2.2 Fault Notification

This section defines a **FaultNotifier** interface that contains methods that allow a Fault Detector or Fault Analyzer to push fault reports to the Fault Notifier. It also defines methods that allow the Replication Manager, a Fault Analyzer or other application object to register as consumers of fault event notifications. The Fault Notifier filters fault reports that it has received from the Fault Detectors, and propagates fault reports to the entities that have registered for such notifications.

27.4.2.3 Fault Analysis

The Fault Analyzer registers with the Fault Notifier as a consumer of fault reports. The Fault Analyzer correlates fault reports and generates condensed fault reports. Because these activities are specific to the application or the environment, the application developer is responsible for the analysis/diagnosis algorithm employed by the Fault Analyzer. The Fault Analyzer may use the Fault Notifier to disseminate its condensed fault reports.

27.4.2.4 Scalability

The fault management specification does not limit the number or arrangement of Fault Detectors in a fault tolerance domain. In a large system spanning many hosts with each host supporting many objects, arranging the Fault Detectors in a hierarchical structure would be more scalable and efficient.

For example, consider a system where all objects at a given location (say, a process) are monitored by a local object-level Fault Detector, as shown in Figure 27-8. The set of object-level Fault Detectors might be monitored by a process-level Fault Detector. The set of process-level Fault Detectors might be monitored by a host-level Fault

Detector. The Replication Manager, or an consumer object created by the Replication Manager, might be implemented to consume either object-level, process-level or host-level fault reports. If it is implemented to consume only object-level fault reports, a Fault Analyzer that translates object-level fault reports into process- or host-level fault reports can be attached to the Fault Notifier.

Monitoring at the process level can be achieved by monitoring a single proxy object in the process. The proxy object would be responsible for ensuring that all of the other objects in the process are alive, and would monitor those objects through the use of application-specific facilities or private Fault Notifier channels provided by the infrastructure.

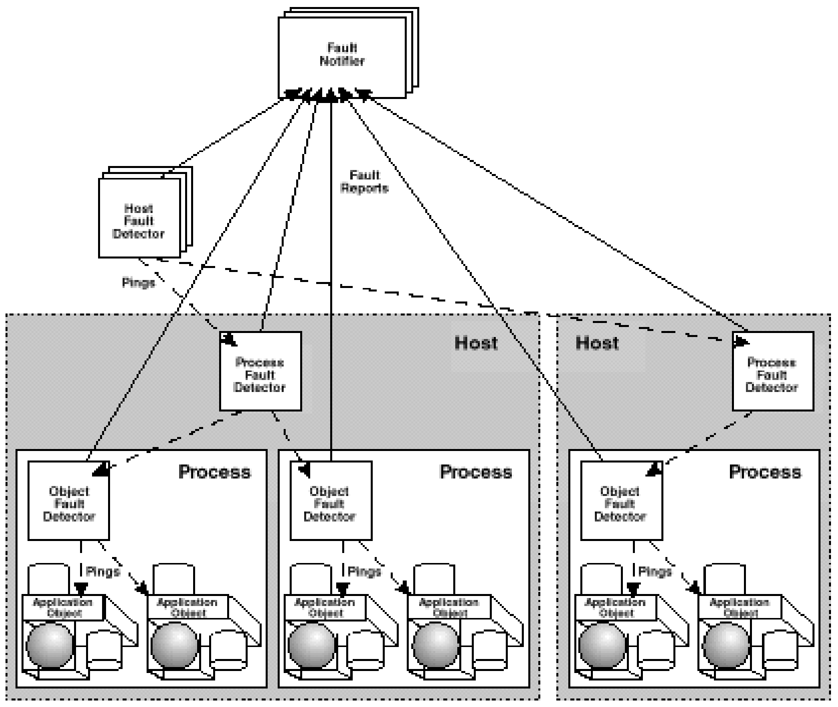

Figure 27-8 Hierarchical Fault Detection.

This example shows the generality of the Fault Tolerance Infrastructure in handling different types of arrangements of Fault Detectors. Other organizations are possible and useful.

27.4.2.5 Deployment of Fault Detectors

Fault Detectors can be as varied as the applications they monitor and, for these diverse applications, Fault Detectors can be deployed in several different ways:

- Statically Deployed Fault Detectors. In an operating environment with a relatively static configuration, location-specific Fault Detectors will typically be created when the Fault Tolerance Infrastructure is installed. For example, these stand-alone Fault Detectors could be implemented as daemon processes that are installed with the Fault Tolerance Infrastructure. These Fault Detectors could be registered in a manner internal to the Fault Tolerance Infrastructure, allowing the infrastructure to include them in every fault-tolerant application within the fault tolerance domain in a transparent manner.
- Infrastructure Created Fault Detectors. The Fault Tolerance Infrastructure may create instances of Fault Detectors to meet the needs of the applications. For example, to implement the MEMB FaultMonitoringGranularity, the Fault Tolerance Infrastructure must create Fault Detectors sufficient to ping every member of the object group. Because these Fault Detectors are created (or, at least, configured) by the Fault Tolerance Infrastructure, their identities need only be known to the infrastructure.
- Application Created Fault Detectors. It might be necessary or advantageous for applications to create their own Fault Detectors. For example, applications might have unique knowledge of their operating environment, such as access to hardware indicators of faults within the operating environment. However, unlike the other types of Fault Detectors, application-created Fault Detectors are not inherently known to the Fault Tolerance Infrastructure. They can propagate their fault information to an application-specific Fault Analyzer through the Fault Notifier provided by the infrastructure. The Fault Analyzer can interpret these application-specific fault reports, generate reports that can be understood by the Replication Manager, and propagate them to the Replication Manager through the Fault Notifier, as shown in Figure 27-8.

27.4.3 Connecting Fault Detectors to Applications

The Fault Notifier provides flexible event-based connection of Fault Detectors to the Replication Manager, Fault Analyzer and other application objects. Fault Detectors, from whatever source, push fault reports onto Fault Notifier channels. The Replication Manager, Fault Analyzer or application objects registers as a consumer of fault reports. The Fault Notifier provides the channel for fault reports in an indirect manner, thus allowing the decoupling of the identity and configuration of the Fault Detectors from the application. The process of connecting the Fault Detectors to the Replication Manager, Fault Analyzer or application objects thus devolves to a process of finding the Fault Notifier with which to register for fault notifications.

Obtaining a reference to the Fault Notifier for a fault tolerance domain involves two steps:

1. Obtain a reference to the Replication Manager, which may be done using **resolve_initial_references()**, as described in Section 27.3.10, “Obtaining the Reference for the Replication Manager,” on page 27-59.
2. Query the Replication Manager for the registered Fault Notifier, which may be done using the **get_fault_notifier()** method of the **ReplicationManager** interface, given in Section 27.3.6, “Replication Manager,” on page 27-42.

The use cases in Section 27.3.11, “Use Cases,” on page 27-59 provide further details.

27.4.4 Pull-Based Monitoring

Based on the MEMB FaultMonitoringGranularity and the PULL FaultMonitoringStyle, the Replication Manager chooses a pull-based Fault Detector to monitor a member of the object group. The pull-based Fault Detector periodically pings the member by invoking the **is_alive()** method of the **PullMonitorable** interface that the member of the object group inherits. The period of the ping is determined by the **FaultMonitoringInterval** for the object group. The pull-based Fault Detector uses the monitoring interval as a hint (in contrast to maintaining the exact value) to optimize monitoring across a number of objects.

27.4.4.1 PULL Fault Monitoring Style

In the PULL FaultMonitoringStyle, the Fault Detector periodically invokes the object to check its liveness; the monitored object responds to these liveness requests. The monitored object must inherit the **PullMonitorable** interface. The Fault Detector invokes the **is_alive()** method of this interface to check the liveness of the object.

Figure 27-9 shows the interactions between the monitored object represented by the **PullMonitorable** interface and the Fault Detector for the PULL FaultMonitoringStyle, and also the interactions with the Fault Notifier and the Replication Manager.

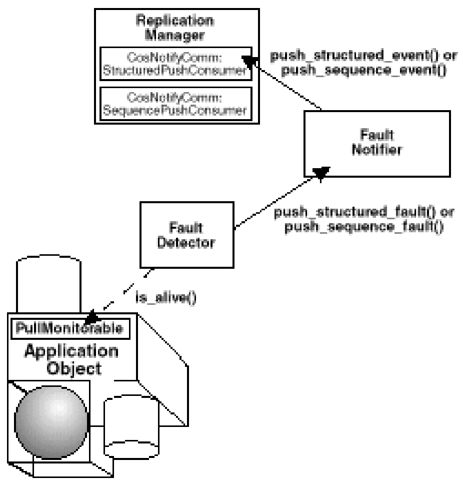
Replication Manager
CosNotifyComm: StructuredPushConsumer
CosNotifyComm: SequencePushConsumer
push_structured_event() or push_sequence_event()
Fault Notifier
push_structured_fault() or push_sequence_fault()
Fault Detector
is_alive()
PullMonitorable
Application Object

Figure 27-9 PULL FaultMonitoringStyle.

27.4.4.2 PullMonitorable Interface

```
module FT {
    interface PullMonitorable {
```

```
                boolean is_alive();
        };
};
```

is_alive

This method 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 (e.g., assertion check) within the method and return false if the test shows that the object is in an inconsistent state.

```
boolean is_alive();
```

Return Value

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

27.4.5 Fault Event Types

Fault reports are conveyed to the Fault Notifier by the Fault Detectors and by the Fault Notifier to the entitites that have registered for such notifications. The Fault Detectors and Fault Notifier use a well-defined event type to convey a given fault event. This specification defines a set of fault event types that are understood by the Fault Tolerance Infrastructure. Vendors or the OMG may extend these fault event types to include other types of fault events.

To align the Fault Tolerant CORBA specification with the CosNotification Service, the fault event types are mandated to be either **CosNotification::StructuredEvent** or **CosNotification::EventBatch** (sequence of **StructuredEvent**). Fault events flow from the Fault Detectors to the Fault Notifier to the consumers according to one of these two formats.

27.4.5.1 ObjectCrashFault

The fault management specification defines one event type: **ObjectCrashFault**. As the name suggests, this event is generated by a Fault Detector when it detects that an object 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 = "ObjectCrashFault";
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 */;
if (all objects at a given location have failed)
    {}  /* do nothing */
else {
        fault_event.filterable_data.length(3);
        fault_event.filterable_data[2].name = "TypeId";
        fault_event.filterable_data[2].value = /* Value of TypeId bundled into any */;
        if (all objects of a given type at a given location have failed)
         {} /* do nothing */
        else {
            fault_event.filterable_data.length(4);
            fault_event.filterable_data[3].name = "ObjectGroupId";
            fault_event.filterable_data[3].value =
                /* Value of ObjectGroupId bundled into any */;
        };
    };
```

The **filterable_data part** of the event body contains the identity of the crashed object as four name-value pairs: the fault tolerance domain identifier, the member's location identifier, the repository identifier and the object group identifier. The Fault Notifier filters events based on the **domain_name**, the **type_name**, and the four identifiers. All other fields of the structured event may be set to null.

The Fault Detector always sets the following fault event fields: **domain_name**, **type_name**, **FTDomainId**, and **Location**. The fault detector may or may not set the TypeId and ObjectGroupId fields with the following interpretations:

- Neither is set if all objects at the given location have failed.
- **TypeId** is set and **ObjectGroupId** is not set if all objects at the given location with the given type have failed.
- Both are set if the member with the given **ObjectGroupId** at the given location has failed.

27.4.6 Fault Notifier

The Fault Notifier takes the fault reports generated by the Fault Detectors or the Fault Analyzers, filters them, and propagates them to entities that have registered for fault notifications, such as the Replication Manager, the Fault Analyzer or other application objects.

The Fault Notifier provides a small subset of the functionality of the **CosNotification** Service. The **CosNotification** Service is complex, and an implementation of the full specification might be difficult to render fault tolerant. The Fault Notifier assumes that the notification channel used for propagating fault reports has the following properties:

- Push-based event communication model
- Support for propagating **CosNotification::StructuredEvent** and **CosNotification::EventBatch** (Sequence of StructuredEvent) types
- Forwarding filter framework at the consumer.

A notification channel that provides the above properties and that can be made fault-tolerant is a good candidate for implementing the Fault Notifier.

The Fault Notifier uses the existing CosNotification StructuredEvent and EventBatch formats, forwarding filter framework, and consumer end interfaces. The default constraint grammar is the same as that supported by the CosNotification Service (see telecom/98-11-01).

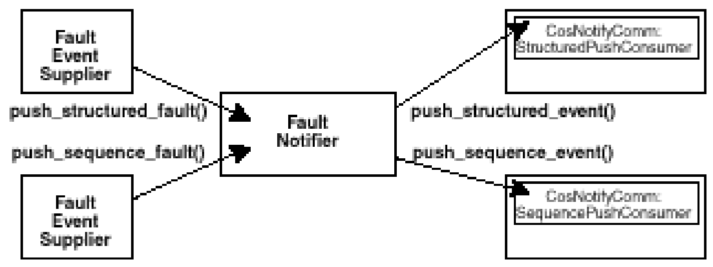

Figure 27-10 Fault Report Propagation through the Fault Notifier.

Figure 27-10 shows the interaction between the Fault Notifier and the fault event suppliers and consumers during fault propagation.

Any fault event supplier (Fault Detector) may obtain the reference to the Fault Notifier and send fault reports to it. It does not need to register explicitly with the Fault Notifier. The **FaultNotifier** interface provides two methods, **push_structured_fault()** and **push_sequence_fault()**, for fault event suppliers to push fault events of the form **CosNotification::StructuredEvent** and **CosNotification::EventBatch** to the Fault Notifier.

A fault event consumer, such as the Replication Manager or a consumer object created by the Replication Manager, must register with the Fault Notifier to receive fault event notifications, as shown in Figure 27-11. The **FaultNotifier** interface provides two methods for registering consumers: **connect_structured_fault_consumer()** for consumers that accepts only structured events and **connect_sequence_fault_consumer()** for consumers that accept a sequence of structured events. A consumer that wishes to receive structured events must support the **CosNotifyComm::StructuredPushConsumer** interface and a consumer that wishes to receive a sequence of structured must support the **CosNotifyComm::SequencePushConsumer** interface.

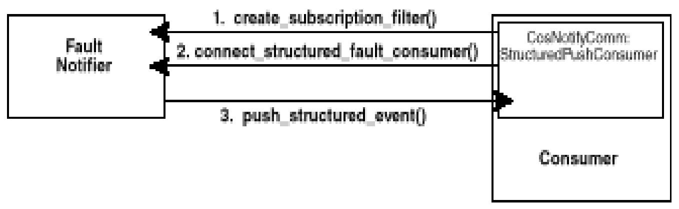
Fault Notifier
1. create_subscription_filter()
2. connect_structured_fault_consumer()
3. push_structured_event()
CosNotifyComm::StructuredPushConsumer
Consumer

Figure 27-11 Connection Setup between the Consumer and the Fault Notifier.

The Fault Notifier propagates all events of a given format to all consumers that accept that format. If a consumer wishes to receive specific fault reports, it must provide filtering information to the Fault Notifier as a parameter to the connect method. The Fault Notifier provides a method, **create_subscription_filter()**, that creates a new object of type **CosNotifyFilter::Filter** and provides the consumer with the handle to that object. The consumer may use the **CosNotifyFilter::Filter::add_constraints()** method to customize the filter with suitable constraints. The consumer passes in the reference to the customized filter object as a parameter to the connect method. While the consumer is connected to the Fault Notifier, it may update the filter constraints using the **CosNotifyFilter::Filter::modify_constraints()** method. Constraints are formulated by the consumer and interpreted by the Fault Notifier in the same manner.

The Fault Tolerant CORBA specification mandates the implementation of a subset of the **CosNotifyFilter::Filter** interface. The required subset deals with manipulation and matching of constraints, and comprises the following attribute and methods: **constraint_grammar**, **add_constraints()**, **modify_constraints()**, **get_constraints()**, **get_all_constraints()**, **remove_all_constraints()**, **destroy()**, and **match_structured()**. For the rest of the methods of the **CosNotifyFilter::Filter** interface, an implementation is allowed to throw a NO_IMPLEMENT exception.

When the consumer is no longer interested in receiving fault reports, it may detach itself from the Fault Notifier using the **disconnect_consumer()** method.

```
module FT {
    interface FaultNotifier {
        typedef unsigned long long ConsumerId;

            void push_structured_fault(
                in CosNotification::StructuredEvent event);

            void push_sequence_fault(
                in CosNotification::EventBatch events);

        CosNotifyFilter::Filter create_subscription_filter (
                in string constraint_grammar)
            raises (CosNotifyFilter::InvalidGrammar);

        ConsumerId connect_structured_fault_consumer(
                in CosNotifyComm::StructuredPushConsumer push_consumer,
                in CosNotifyFilter::Filter filter) ;

        ConsumerId connect_sequence_fault_consumer(
                in CosNotifyComm::SequencePushConsumer push_consumer,
                in CosNotifyFilter::Filter filter);

            void disconnect_consumer(in ConsumerId connection)
            raises(CosEventComm::Disconnected);
    };
};
```

27.4.6.1 Identifiers

typedef unsigned long long ConsumerId;

The identifier used to identify the consumer of notifications uniquely within the Fault Notifier.

27.4.6.2 Operations

push_structured_fault

The supplier of a fault report creates a structured event containing the fault report and invokes this method with the structured event as an in parameter. The Fault Notifier then pushes a fault notification to the consumers that have registered for such notifications.

```
void push_structured_fault(in CosNotification::StructuredEvent event);
```

Parameters

event	The fault event that is to be delivered to the consumer.

push_sequence_fault

The supplier of a fault report creates a sequence of structured event containing the fault reports and invokes this method with the sequence of structured event as an in parameter. The Fault Notifier then pushes a fault notification to the consumers that have registered for such notifications.

```
void push_sequence_fault(in CosNotification::EventBatch events);
```

Parameters

event	The fault event that is to be delivered to the consumer.

create_subscription_filter

This method creates a new forwarding filter object of type **CosNotifyFilter::Filter**. It takes as an in parameter a string identifying the constraint grammar within which the constraints on the newly created filter are defined and returns the reference to the filter object.

```
CosNotifyFilter::Filter create_subscription_filter (in string constraint_grammar)
        raises (CosNotifyFilter::InvalidGrammar);
```

Parameters

constraint_grammar	A string used to identify the constraint grammar for creating the new filter.

Return Value

The reference to the filter object created by the Fault Notifier.

Raises

CosNotifyFilter::InvalidGrammar if the Fault Notifier does not support the grammar identified by the in parameter.

connect_structured_fault_consumer

This method accepts as an in parameter the reference to a consumer that wishes to receive structured events from the Fault Notifier and returns an identifier that uniquely identifies the consumer within the context of the Fault Notifier. The consumer must use this identifier in all of its subsequent interactions with the Fault Notifier. The method establishes a logical connection between the Fault Notifier and the consumer, and allows the Fault Notifier to push fault events to the consumer, using the **push_structured_event()** method of the **CosNotifyComm::StructuredPushConsumer** interface.

The consumer may tailor the types of fault events that it wishes to receive by supplying a filter as the second parameter. It creates the filter using the **create_subscription_filter()** method of the **FaultNotifier** interface, and manipulates the constraints on the filter using the **add_constraints()** and **modify_constraints()** methods of the **CosNotifyFilter::Filter** interface.

```
ConsumerId connect_structured_fault_consumer(
        in CosNotifyComm::StructuredPushConsumer push_consumer,
        in CosNotifyFilter::Filter filter) ;
```

Parameters

push_consumer	The reference to the consumer object that is registering for fault notifications.
filter	The reference to the filter that the Fault Notifier must use to restrict the set of events that are to be delivered to the consumer.

Return Value

An identifier that uniquely identifies the consumer within the context of the Fault Notifier and is used by the consumer in subsequent interactions with the Fault Notifier.

connect_sequence_fault_consumer

This method accepts as an in parameter the reference to a consumer that wishes to accept a sequence of structured events from the Fault Notifier and returns an identifier that uniquely identifies the consumer within the context of the Fault Notifier. The consumer must use this identifier in all of its subsequent interactions with the Fault Notifier. The method establishes a logical connection between the Fault Notifier and the consumer, and allows the Fault Notifier to push fault events to the consumer using the **push_sequence_event()** method of the **CosNotifyComm::SequencePushConsumer** interface.

The consumer may tailor the types of fault events that it wishes to receive by supplying a filter as the second parameter. It creates the filter using the **create_subscription_filter()** method of the **FaultNotifier** interface, and manipulates the constraints on the filter using the **add_constraints()** and **modify_constraints()** methods of the **CosNotifyFilter::Filter** interface.

```
ConsumerId connect_sequence_fault_consumer(
            in CosNotifyComm::SequencePushConsumer push_consumer,
            in CosNotifyFilter::Filter filter) ;
```

Parameters

push_consumer	The reference to the consumer object that is registering for fault notifications.
filter	The reference to the filter that the Fault Notifier must use to restrict the set of events to be delivered to the consumer.

Return Value

An identifier that uniquely identifies the consumer within the context of the Fault Notifier and that is used by the consumer in subsequent interactions with the Fault Notifier.

disconnect_consumer

This method is invoked by the consumer to disconnect itself from the Fault Notifier. The method takes as an in parameter the ConsumerId identifying the disconnecting consumer.

```
void disconnect_consumer(in ConsumerId connection)
      raises(CosEventComm::Disconnected);
```

Parameters

connection	The ConsumerId identifying the particular consumer that wishes to be disconnected.

Raises

CosEventComm::Disconnected if the Fault Notifier is not currently connected to any consumer ientifier by the given ConsumerId.

27.4.6.3 Filtering

Filtering is done through the **CosNotification** forwarding filter framework. The **CosNotifyFilter::Filter** interface is a mandatory part of the Fault Tolerant CORBA specification. The filtering constraints must be specified using the Extended Trader Constraint Language in the CosNotification Service (see telecom/98-11-01).

Because Location is of type **CosNaming::Name**, a location can be described using a hierarchical location scheme. For example, an object “objA” located in process “procB” on host “hostC” can be described as follows:

```
Location object_location;
object_location.length(3);
object_location[0].id = "hostC";
object_location[0].kind = "hostname";
object_location[1].id = "procB";
object_location[1].kind = "processname";
object_location[2].id = "objA";
object_location[2].kind = "objectname";
```

To facilitate hierarchical fault detection and reporting, the Fault Detector may omit some trailing Location entries. For example, if all objects on a host fail, then a Fault Detector may send a fault report with only the leading Location entry, which identifies the failed host.

The Fault Notifier may also filter events based on a subset of the Location entries. For example, if a consumer of fault events wishes to subscribe to notifications of faults of type **ObjectCrashFault** on a particular host, the filtering selects faults based on the leading entry of Location, which identifies the host.

The Extended Trader Constraint Language is used to filter fault events, as illustrated below.

For example, to register for all fault events in ftdom0 on hostC, use the filter string "$event_type.domain_name == ‘FT_CORBA’ and $event_type.type_name == ‘ObjectCrashFault’ and $FTDomainId == ‘ftdom0’ and $Location[0].id == ‘hostC’".

To register for fault events for a member of an object group, identified by (ftdom0, group1, type2, hostC, procB), where the object itself crashed or the process containing the object crashed or the host supporting the process crashed, use the filter string "$event_type.domain_name == ‘FT_CORBA’ and $event_type.type_name == ‘ObjectCrashFault’ and $FTDomainId == ‘ftdom0’ and (not exists $ObjectGroupId or $ObjectGroupId == ‘group1’) and (not exists $TypeId or $TypeId == ‘type2’) and $Location[0].id == ‘hostC’ and (not exists $Location[1] or $Location[1].id == ‘procB’)".

27.4.6.4 Mapping of the Fault Notifier to the CosNotification Service

This section is intended as an informational, rather than a mandatory, part of the specification. It is intended for vendors that want to use the **CosNotification** service, in place of the **FaultNotifier** interface that has been defined in this specification. Such a vendor must use an implementation of the **CosNotification** service that can be rendered fault-tolerant and that is compatible with the rest of the Fault Tolerance Infrastructure. The six methods of the **FaultNotifier** interface map directly or indirectly to one or more of the methods of the **CosNotification** service.

Initialization

The Fault Notifier first creates a notification channel and registers itself both as a structured event supplier and a sequence of structured event supplier with the notification channel. To register itself as a supplier of structured events, the Fault Notifier goes through the following steps:

1. It invokes **CosNotifyChannelAdmin::EventChannel::default_supplier_admin()** and gets the reference to the **CosNotifyChannelAdmin::SupplierAdmin** interface.
2. It invokes **obtain_notification_push_consumer()** on the **SupplierAdmin** interface and gets a reference to the **CosNotifyChannelAdmin::ProxyConsumer** interface, which it narrows to **CosNotifyChannelAdmin::StructuredProxyPushConsumer**.
3. It invokes **connect_structured_push_supplier()** on the **StructuredProxyPushConsumer** to connect itself as a supplier of structured events.

The Fault Notifier follows similar steps to register itself as a supplier of a sequence of structured events.

Supplier End Methods

The supplier end methods **push_structured_fault()** and **push_sequence_fault()** map to **CosNotifyComm::StructuredProxyPushConsumer::push_structured_event()** and **CosNotifyComm::SequenceProxyPushConsumer::push_sequence_event()**.

Consumer End Methods

Consumers, such as the Replication Manager or a consumer object created by the Replication Manager, connect to the Fault Notifier through the **connect_structured_fault_consumer()** and **connect_sequence_fault_consumer()** methods. Before the consumers can connect, they must create the filter they want for the connection by invoking the **create_subscription_filter()** method. The **create_subscription_filter()** method is implemented by the Fault Notifier using a sequence of invocations on the notification channel. The Fault Notifier obtains the reference to the **CosNotifyFilter::FilterFactory** interface by invoking the **default_filter_factory()** method on the **CosNotifyChannelAdmin::EventChannel** interface. It asks the filter factory to create a new filter with a given constraint grammar by invoking the **CosNotifyFilter::FilterFactory::create_filter()** method and returns the reference to the newly created filter as the return value of **create_subscription_filter()**.

In response to the **connect_structured_fault_consumer()** invocation, the Fault Notifier goes through the following sequence of steps to set up the connection between the consumer and the notification channel.

1. It invokes **CosNotifyChannelAdmin::EventChannel::default_consumer_admin()** and gets the reference to the **CosNotifyChannelAdmin::ConsumerAdmin** interface.
2. It invokes **obtain_notification_push_supplier()** on the **ConsumerAdmin** and gets a reference to the **CosNotifyChannelAdmin::ProxySupplier** interface which it narrows to **CosNotifyChannelAdmin::StructuredProxyPushSupplier**.
3. It invokes **connect_structured_push_consumer()** on the **StructuredProxyPushSupplier** and passes it the reference to the connecting consumer. This sets up a connection capable of propagating structured fault events between the notification channel and the push consumer.
4. To set up the filtering, the Fault Notifier invokes **add_filter()** on the **StructuredProxyPushSupplier,** with an in parameter that is the filter that was passed to the Fault Notifier in the invocation of **connect_structured_fault_consumer()**.

27.4.7 Use Cases

27.4.7.1 The Fault Detector as a Fault Notification Supplier

1. The Replication Manager wishes to monitor an object O1 with reference O1_ref. The object belongs to the fault tolerance domain “acme.com” and object group “1” and location “object_location.” Based on the PULL FaultMonitoringStyle and the location of the object, the Replication Manager chooses a pull-based Fault Detector and informs it to start monitoring the object with the value of the FaultMonitoringInterval given as a property.
2. The pull-based Fault Detector periodically invokes **is_alive()** on O1_ref.
3. If Object O1 fails to respond to the **is_alive()** messages of the Fault Detector, the Fault Detector may declare the object to have crashed. It then takes the following actions:
   - It creates a StructuredEvent data structure with the following data.

```
Location object_location;
object_location.length(1);
object_location[0].id = "myhost.acme.com";
object_location[0].kind = "hostname";
CosNotification::StructuredEvent fault_event;
fault_event.header.fixed_header.event_type.domain_name = "FT_CORBA";
fault_event.header.fixed_header.event_type.type_name = "ObjectCrashFault";
fault_event.filterable_data.length(4);
fault_event.filterable_data[0].name = "FTDomainId";
fault_event.filterable_data[0].value <<= "acme.com";
fault_event.filterable_data[1].name = "Location";
fault_event.filterable_data[1].value <<= object_location;
fault_event.filterable_data[2].name = “TypeId”;
fault_event.filterable_data[2].value <<= object_type;
fault_event.filterable_data[3].name = "ObjectGroupId";
fault_event.filterable_data[3].value <<= 1;
```

- It invokes **push_structured_event(fault_event)** on the Fault Notifier.

27.4.7.2 The Replication Manager as a Fault Notification Consumer

1. The Replication Manager wishes to be notified when object O1 crashes.
2. It invokes **create_subscription_filter()** on the Fault Notifier.The Fault Notifier returns the reference to the newly created filter object.
3. The Replication Manager creates a **CosNotifyFilter::ConstraintExpSeq** with the following data:

```
CosNotifyFilter::ConstraintExpSeq_var my_exp_seq;
my_exp_seq.length(1);
my_exp_seq[0].event_types.length(1);
my_exp_seq[0].event_types[0].domain_name = "FT_CORBA";
my_exp_seq[0].event_types[0].type_name = "ObjectCrashFault";
my_exp_seq[0].constraint_expr = "$FTDomainId == ‘acme.com’
                                           and $ObjectGroupId == 1
                                           and $Location[0].id ==‘myhost.acme.com’";
```

4. The Replication Manager invokes **add_constraints(my_exp_seq)** on the filter object returned in Step 2. The above constraints allow the Replication Manager to register for **ObjectCrashFault** of a member of object group 1 occurring on host “myhost.acme.com”.
5. Once the filter creation and initialization are complete, the Replication Manager invokes **connect_structured_fault_consumer()** with a push consumer reference and the filter reference as the in parameters. The Fault Notifier returns a consumer identifier (**c_id**) to the Replication Manager. This completes the registration process.
6. When the Replication Manager is no longer interested in fault reports for O1, it invokes **modify_constraints()** on the filter object with suitable constraint values.
7. If the Replication Manager does not wish to receive any more notifications, it disconnects from the Fault Notifier by invoking **disconnect_consumer(c_id)** on it.

27.5 Logging & Recovery Management

27.5.1 Overview

The Fault Tolerance Infrastructure includes Logging and Recovery Management Mechanisms that support the infrastructure-controlled ConsistencyStyle. During normal operation, the Logging Mechanism records the state and actions of the primary member of a passively replicated object group in a log. After a fault, the Recovery

Mechanism retrieves these records from the log and uses them to restore the state of a backup member of the object group, so that it can continue the service provided by the primary member that failed. The Logging and Recovery Mechanisms are also used to activate a new member of an actively replicated object group. No interfaces are defined for the Logging and Recovery Mechanisms because these mechanisms are never invoked directly by the application program.

This section defines two interfaces that objects of the application program inherit: **Checkpointable** and **Updateable**. An application object that needs to have its state logged and restored must inherit the **Checkpointable** interface. In addition, it may inherit the **Updateable** interface, which allows state changes to be logged and restored incrementally.

27.5.2 Logging Mechanism

During normal operation, the Logging Mechanism records the state and actions of a member of an object group in a log, as shown in Figure 5-1. The state and actions correspond to messages sent and received by the member of the object group. Conceptually, the Fault Tolerance Infrastructure maintains a distinct log for each object group, although it may record the logs for many object groups within the same physical log. The log may be distributed, in which case it is maintained in local volatile storage at each member of the object group that is the destination of the message. The distributed logging strategy typically employs a reliable totally-ordered multicast protocol to deliver the messages to all of the members of the object group. Alternatively, particularly for passively replicated object groups, the log may be written to shared stable storage by the primary member of the object group that is the source of the message. To be sound, the shared logging strategy requires that each message is forced to the log on stable storage before it is transmitted, which may have an adverse effect on performance.

The format of the log is not specified in this specification. Typically, the information recorded in the log consists of request and reply messages, and states and updates in the form of **get_state()** and **get_update()** request and reply messages, as shown in Figure 27-12 on page 27-83. The log must preserve the order in which messages were received by the members of the object group, so that they can be replayed in the correct order during recovery. States and updates must be positioned logically in the message sequence at the point at which they were requested by the **get_state()** or **get_update()** request message, even though the state or update may be contained in a reply message that is sent at a later time. A complete state consists of the **get_state()** request message and the reply to that request. A complete update is defined similarly.

To conserve memory, the Logging Mechanism must prune the log of records that the Recovery Mechanism will not subsequently require for recovery. Thus, if the log contains a complete state, the Logging Mechanism can discard all log records prior to the **get_state()** request message for that state. Similarly, if a log contains a complete update, the Logging Mechanism can discard all request and reply messages, other than those associated with the logging of a state or update, that precedes the **get_update()** request message for that update. If, however, a request contains an **FT_REQUEST** service context, which defines an expiration time for the request, the request and its matching reply must be retained until that expiration time.

27.5.3 Recovery Mechanism

The Recovery Mechanism sets the state of a member, either after a fault when a backup member of an object group is promoted to the primary member, or alternatively when a new member is introduced into an object group. The Recovery Mechanism processes the log and applies messages from the log to the member to bring that member to the correct current state, so that it can start to process messages normally, as shown in Figure 27-12 on page 27-83.

The messages in the log are not necessarily in the order required for recovery. The Recovery Mechanism processes the log, discarding irrelevant messages to form a complete log. A complete log for an object group contains:

- The most recent complete state in the log. Prior complete states are ignored and can be discarded from the log. Subsequent incomplete states are ignored but are retained in the log so that they can be completed.
- All complete updates that occur after the most recent complete state. Complete updates that occur prior to the most recent complete state are ignored and can be discarded from the log. Subsequent incomplete updates are ignored but are retained in the log so that they can be completed.

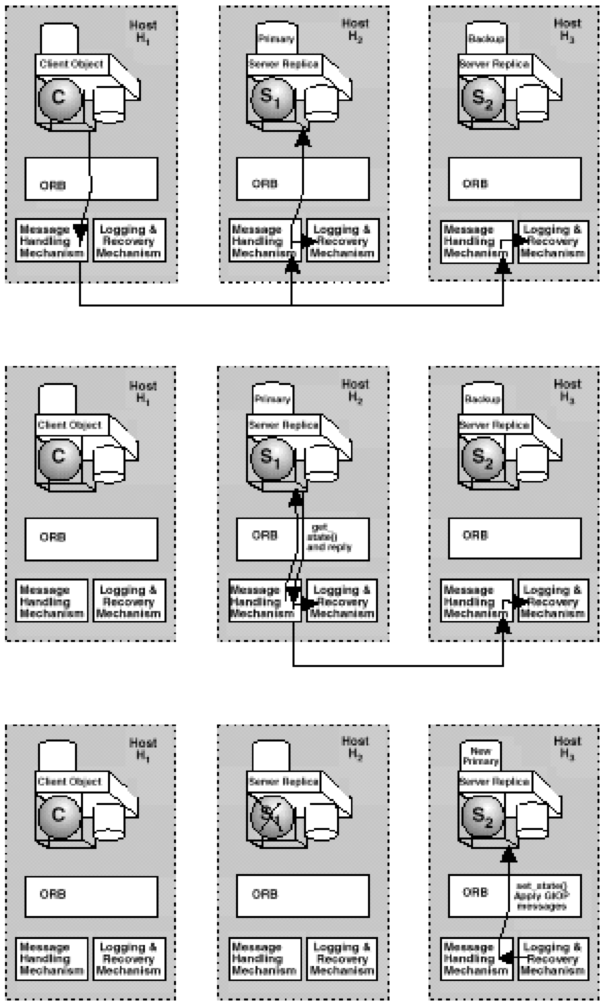

Figure 27-12 Operation of the Logging and Recovery Mechanisms for a server object group having the WARM_PASSIVE ReplicationStyle, during normal operation, during the recording of a checkpoint, and during recovery

- All request and reply messages that occur in the log after the most recent complete state and after the most recent complete update, if present. Request and reply messages are ignored and can be discarded from the log if they occur before the complete state or complete update and if they are not the most recent request and reply messages in the sequence of request and reply messages for a client object group's invocations of this object group.

For a backup member of an object group with the COLD_PASSIVE ReplicationStyle that is being promoted to primary member, or for a new member of an object group with the ACTIVE **ReplicationStyle**, the Recovery Mechanism must apply the entire complete log to the member.

For a backup member of an object group with the WARM_PASSIVE ReplicationStyle that is being promoted to primary member, the member has already received states and updates during normal operation. The Recovery Mechanism applies to the member, only messages in the complete log that follow the most recent state or update applied to the member during normal operation.

For a new backup member of an object group with the WARM_PASSIVE **ReplicationStyle**, the Recovery Mechanism applies only the state and update messages in the complete log to the member.

27.5.4 Checkpointable and Updateable Interfaces

An application object inherits the **Checkpointable** interface, which provides **get_state()** and **set_state()** methods, to enable the Logging and Recovery Mechanisms to record and restore its state. The Logging Mechanism obtains the value of the **CheckpointInterval** from the Property Manager, which determines the interval between successive invocations of the **get_state()** method.

An application object may also inherit the **Updateable** interface, which provides **get_update()** and **set_update()** methods, to enable the Logging and Recovery Mechanisms to record and restore updates. An update is the set of changes in the state of an object since the most recent invocation of **get_state()** or **get_update()**.

The Logging Mechanism invokes the **get_state()** method on a member of an object group to obtain its state. In addition, for the WARM_PASSIVE **ReplicationStyle**, the Logging Mechanism invokes the **get_state()** method on the primary member to obtain the state needed to update the backup members in order to speed up the failover process in case the primary fails. The Recovery Mechanism invokes the **set_state()** method on the new or recovering member of the object group, and on the backups for the WARM_PASSIVE **ReplicationStyle**.

The Logging Mechanism invokes the **get_update()** method on a member of an object group to obtain data that represents the change (delta) between the previous state and the current state. The "previous" state is the state at the moment of the most recent invocation of **get_state()** or **get_update()**. The state of the backup is typically updated using the most recent state plus the following updates. The Recovery Mechanism invokes the **set_update()** method on the new or recovering member of the object group, and on the backups for the WARM_PASSIVE **ReplicationStyle**.

```
module FT {
    typedef sequence<octet> State;

    exception NoStateAvailable {};
    exception InvalidState {};
    exception NoUpdateAvailable {};
    exception InvalidUpdate {};
```

```
        interface Checkpointable {
                State  get_state()
                        raises(NoStateAvailable);

                void set_state(in State s)
                        raises(InvalidState);
        };

        interface Updateable : Checkpointable {
                 State  get_update()
                           raises(NoUpdateAvailable);

                void set_update(in State s)
                        raises(InvalidUpdate);
        };
};
```

27.5.4.1 Identifiers

typedef sequence<octet> State;

The state or partial state (update) of an object.

27.5.4.2 Exceptions

NoStateAvailable {};	This exception is thrown if the state of the object is not available.
InvalidState {};	This exception is thrown if the state being supplied to the object is not a valid state for the object. The Fault Tolerance Infrastructure then assumes that the object has failed.
NoUpdateAvailable {};	This exception is thrown if an update for the object is not available.
InvalidUpdate {};	This exception is thrown if the update being supplied to the object is not a valid update for the object. The Fault Tolerance Infrastructue then assumes that the object has failed.

27.5.4.3 Operations

get_state

This method obtains the state of the application object on which it is invoked. The method is invoked by the Logging Mechanism. The **CheckpointInterval** obtained from the Property Manager determines the interval between invocations of **get_state()**.

When the Logging Mechanism invokes **get_state()**, the application object returns the state. For each retrieval of a state, the Logging Mechanism invokes **get_state()** only once, and the state that is returned is the state at the time **get_state()** is invoked.

```
State  get_state()
      raises(NoStateAvailable);
```

Return Value

The state of the application object on which the method is invoked.

Raises

NoStateAvailable if the state is not available.

set_state

This method sets the state of the application object on which it is invoked. The method is invoked by the Recovery Mechanism. When the Recovery Mechanism invokes **set_state()**, it transfers the state to the application object.

```
void set_state(in State s)
      raises(InvalidState);
```

Parameters

s	The state to be used to set the state of the application object on which the method is invoked.

Raises

InvalidState if the parameter a is not a valid state. If the exception is raised, the Fault Tolerance Infrastructure assumes that the application object has failed.

get_update

This method obtains an update from the application object on which it is invoked. The **get_update()** method is invoked by the Logging Mechanism.

When the the Logging Mechanism invokes **get_update()**, the application object returns the update. For each retrieval of an update, the Logging Mechanism invokes **get_update()** only once, and the update that is returned is the update at the time **get_update()** is invoked.

```
State  get_update()
      raises(NoUpdateAvailable);
```

Return Value

An update for the application object on which the method is invoked.

Exception

NoUpdateAvailable if an update is not available.

27.5.4.4 *set_update*

This method applies an update to the application object on which it is invoked. The method is invoked by the Recovery Mechanism. When the Recovery Mechanism invokes **set_update()**, it transfers the update to the application object.

```
void set_update(in State s)
    raises(InvalidUpdate);
```

Parameters

s	The update to be applied to the application object on which the method is invoked. If the exception is raised, the Fault Tolerance Infrastructure assumes that the application object has failed.

Exception

InvalidUpdate if the parameter a is not a valid update.

27.5.5 *Use Case*

27.5.5.1 *Infrastructure-Controlled Consistency Style*

For the COLD_PASSIVE **ReplicationStyle** and the PULL **FaultMonitoringStyle**, the interactions between the various components of the Fault Tolerance Infrastructure are typically as follows:

1. The Pull Monitor invokes **is_alive()** on the primary member of the object group and the primary responds.
2. The primary fails.
3. The Pull Monitor invokes **is_alive()** on the primary member of the object group and the primary does not respond.
4. The Pull Monitor incurs a timeout and reports to the Fault Notifier that the primary is faulty.
5. The Fault Notifier notifies the Replication Manager that the primary is faulty.
6. The Replication Manager determines the object group containing the primary, and the Replication Style of the object group.
7. The Replication Manager invokes the Fault Tolerance Infrastructure to remove the failed primary from the object group.

8. If the number of members of the object group is now less than the minimum number of replicas for this object group, the Replication Manager initiates the creation of a new member of the object group.
9. If the backup is not yet loaded, the Replication Manager invokes a method of the Fault Tolerance Infrastructure to load the backup.
10. The Replication Manager then invokes a method of the Fault Tolerance Infrastructure to set the new primary for the object group.
11. The Replication Manager invokes a method of the Recovery Mechanism to activate the new primary
12. The Recovery Mechanism accesses the log and extracts the most recent state message for the previous primary and the subsequent request and reply messages.
13. The Recovery Mechanism invokes **set_state()** from the request and reply messages on the new primary.
14. The Recovery Mechanism returns a reply to the Replication Manager's invocation of activate.
15. The Replication Manager invokes the Pull Monitor to start monitoring the new primary.

27.6 Compliance

27.6.1 Fault Tolerant CORBA Passive Replication Compliance Point

This compliance point requires support of all specifications defined previously. However, the implementation of these specifications need only support the semantics for the STATELESS, COLD_PASSIVE, and WARM_PASSIVE values of the **ReplicationStyle** property.

27.6.2 Fault Tolerant CORBA Active Replication Compliance Point

This compliance point requires support of all specifications defined previously. However, the implementation of these specifications need only support the semantics for the STATELESS and ACTIVE values of the **ReplicationStyle** property.

Glossary

Active Replication	All of the members of an object group independently execute the methods invoked on the object, so that if a fault prevents one replica from operating correctly, the other replicas will produce the required results without the delay incurred by recovery.
Active Replication with Voting	Active replication where the requests (replies) from the members of a client (server) object group are voted, and are delivered to the members of the server (client) object group only if a majority of the requests (replies) are identical.
Application-Controlled Consistency	A ConsistencyStyle in which the application is responsible for checkpointing, logging, activation and recovery, and for maintaining whatever kind of consistency is appropriate for the application.
Application-Controlled Membership	A MembershipStyle in which the application, or an application-level manager, can create a member of the object group and then invoke the **add_member()** method of the **ObjectGroupManager** interface to cause the Replication Manager to add the member to the group. Alternatively, the application can invoke the **create_member()** method of the **ObjectGroupManager** interface to cause the Replication Manager to create the member and add it to the object group. The application is responsible for enforcing the InitialNumberReplicas and MinimumNumberReplicas properties.
Backup Member	In passive replication, a member of an object group that does not execute the methods invoked on the object group but is available to assume the role of the primary member in the event of a fault.
Byzantine Fault	A form of commission fault that occurs when an object or host generates incorrect results maliciously.
Causal Order	Causal order ensures that if a multicast message m1 could have caused, possibly indirectly, a message m2 then no object receives m2 before it receives m1. The *causally precedes* relation is the transitive closure of:
	• If message m1 is delivered to object replica O before O sends message m2, then m1 causally precedes m2. • If object replica O sends message m1 before message m2, then m1 causally precedes m2. • If both m1 and m2 are delivered to object replica O, and m1 causally precedes m2, then m1 is delivered to O before m2.
Checkpoint	A snapshot of the state of an object.

Checkpoint Interval	An interval of time (in seconds and nanoseconds) between writing the full state of an object to a log.
Cold Passive Replication	A form of passive replication in which only one replica, the primary replica, in the object group executes the methods invoked on the object. The state of the primary replica is extracted from the log and is loaded into the backup replica when needed for recovery.
Commission Fault	A commission fault occurs when an object or host generates incorrect results. Commission faults must be handled by active replication with majority voting.
ConsistencyStyle	The value of the ConsistencyStyle is either CONS_INF_CTRL or CONS_APP_CTRL.
Distributed Logging	A logging strategy in which a co-located log is maintained for each replica of an object.
Duplicates	Duplicate requests and duplicate replies can arise in active replication and also in passive replication when the primary fails and a new primary is introduced. To maintain exactly once semantics and strong replica consistency, the Fault Tolerance Infrastructure provides mechanisms to detect and suppress duplicates.
Failure	A failure is the event of a system's generating a result that does not satisfy the system specification or not generating a result that is required by the system specification. A failure is defined by the system specification, without reference to any enclosing system of which the system is a component.
Fault	A fault is behavior of a component of a system that causes incorrect behavior of the system. A fault is the external manifestation of a failure of the component.
Fault Analyzer	A component of the Fault Tolerance Infrastructure that registers for fault notifications and aggregates multiple related fault notifications into a single fault report.
Fault Containment Region	One or more locations that can be affected by a single fault. Each member of an object group is assigned to a different fault containment region to ensure that, if one member incurs a fault, the other members are not affected.
Fault Monitor	A component of the system, also known as a Fault Detector, that monitors the occurrence of faults in other entities, such as objects, hosts, processes and networks. Fault detectors are typically based on timeouts and are unreliable (inaccurate) because they cannot determine whether an entity has failed or is merely slow.

FaultMonitoringGranularity	The value of the FaultMonitoringGranularity of an object group is either MEMB, LOC, or LOC_AND_TYPE. The FaultMonitoringGranularity provides a means of scalably monitoring the members of many object groups.
FaultMonitoringIntervalAndTimeout	The value of the FaultMonitoringIntervalAndTimeout is a structure that contains an interval of time between successive pings of an object, and the time allowed for subsequent responses from the object to determine whether it is faulty.
FaultMonitoringStyle	The value of the FaultMonitoringStyle is either PULL, PUSH or NOT_MONITORED.
Fault Tolerance	The ability to provide continuous service, unperturbed by the presence of faults. In contrast, with high availability, existing operations can be disrupted by a fault but subsequent new operations, or retried existing operations, are serviced.
Fault Tolerance Domain	For scalability, large applications are divided into multiple fault tolerance domains, each managed by a single Replication Manager. The members of an object group 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.
Fault Transparency	A server object group is fault transparent to a client object if, in the presence of a faulty server replica, the server object group interacts with the client object as if there were no faults.
Gateway	A gateway provides access into a fault tolerance domain for objects outside that domain, and also provides protocol conversion between the IIOP protocol used outside the fault tolerance domain and the group communication protocol used inside that domain.
GenericFactory	An interface of the Replication Manager that creates object groups, as well as individual members of object groups.
Group Communication Protocol	A protocol that provides communication between object groups, typically multicasting, reliable delivery, causal ordering, total ordering, group membership and virtual synchrony.
Group Membership	The set of members of a group, which may change dynamically in time, as members fail and are removed from the group and as new and recovered members are added.
FT_GROUP_VERSION Service Context	A service context, included in a request message, that allows a server to determine whether the client is using an obsolete object group reference and, if so, to return a LOCATION_FORWARD_PERM response that contains the most recent object group reference for the server object group.

HEARTBEAT_POLICY	A client-side policy that allows a client to request heartbeating to determine that its connection to a server has failed.
HEARTBEAT_ENABLED_POLICY	A server-side policy that allows a client to determine that its connection to a server has failed.
Incremental State Transfer	A form of state transfer that is used for transferring large states of an object in fragments.
Infrastructure-Controlled Consistency	A ConsistencyStyle in which the Fault Tolerance Infrastructure is responsible for checkpointing, logging, activation and recovery and for maintaining Strong Replica Consistency.
Infrastructure-Controlled Membership	A MembershipStyle in which the application directs the Replication Manager to create the object group and the Replication Manager invokes the individual factories, for the appropriate locations, to create the members of the object group both initially to satisfy the InitialNumberReplicas property and also after the loss of a member because of a fault to satisfy the MinimumNumberReplicas property.
InitialNumberReplicas	The InitialNumberReplicas property of an object group specifies the number of replicas of the object to be created when the object group is first created.
Location	A set of hosts that form a single fault containment region. Members of object groups are created at different locations.
Log	A record of messages and object states that is created to ensure that recovery is possible after a fault.
Logging Mechanism	A component of the Fault Tolerance Infrastructure that records all of the actions of an object group in a log.
MembershipStyle	The value of the MembershipStyle of an object group is either MEMB_INF_CTRL or MEMB_APP_CTRL.
Message Handling Mechanism	A component of the Fault Tolerance Infrastructure that ensures that GIOP messages addressed to object groups are delivered to the appropriate members of those groups. It detects and suppresses duplicate messages, passes messages to the Logging Mechanism to put into the log, and applies to the objects messages that the Recovery Mechanism has retrieved from the log.
MinimumNumberReplicas	The MinimumNumberReplicas property of an object group specifies the smallest number of replicas of the object needed to maintain the desired fault tolerance. The application or the Replication Manager creates additional replicas of the object to ensure that the number of replicas does not fall below the specified minimum number.

Multicasting	For replicated client and server objects, messages are originated by a client (server) within a client (server) object group and are multicast to the client and server object groups. Messages are delivered to the members of both the client and server object groups to facilitate the detection and suppression of duplicates.
Object Group	A set of member objects, each of which implements the same set of interfaces and has the same implementation code.
ObjectGroupManager	An interface of the Replication Manager that contains methods 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, and setting the primary member of a passively replicated object group.
Object Group Reference	An interoperable object reference that contains multiple TAG_INTERNET_IOP profiles that represent primary and backup members of a passively replicated object group or that represent gateways. All of the TAG_INTERNET_IOP profiles contain a TAG_FT_GROUP component that contains the fault tolerance domain identifier, object group identifier, and object group reference version number for the server object group. If the profiles are those of members of a passively replicated server object group, then one of the profiles contains the TAG_FT_PRIMARY component for the profile that addresses the primary member of the server object group.
Passive Replication	Only the primary member of an object group executes the methods that have been invoked on the object group. The object group contains additional backup replicas.
Primary Member	In passive replication, the member of an object group that executes the methods invoked on the object group.
Property Manager	An interface of the Replication Manager that contains methods for setting and getting the fault tolerance properties.
Pull Monitor	A Fault Monitor that interrogates the monitored object periodically to determine whether it is alive.
Push Monitor	A Fault Monitor to which the monitored object periodically reports that it is alive.
Recovery	The restoration of the state of a member of an object group so that it can continue the operation of the object group.

Recovery Mechanism	A component of the Fault Tolerance Infrastructure that sets the state of a member of an object group, either when a backup member is promoted to be the primary member after a fault occurs, or alternatively when a new member is introduced into the group.
Reliable Delivery	Every message addressed to a group, or originated by a group, is delivered to every member of the group, except for members suspected of being faulty.
Replica Determinism	Replica determinism requires that two or more members of an object group, when presented with the same sequence of requests and replies, behave in exactly the same manner.
Replication	The fundamental technique used in building fault-tolerant systems.
Replication Manager	A component of the Fault Tolerance Infrastructure that provides access to the Fault Notifier and that inherits three interfaces: **PropertyManager**, **GenericFactory** and **ObjectGroupManager**. Logically, there is one Replication Manager per fault tolerance domain. The Replication Manager interacts with the Fault Monitors and Fault Notifier, and also with the Logging and Recovery Mechanisms of the Fault Tolerance Infrastructure.
ReplicationStyle	The value of the ReplicationStyle of an object group is either STATELESS, COLD_PASSIVE, WARM_PASSIVE, ACTIVE, or ACTIVE_WITH_VOTING.
Replication Transparency	A client object is unaware that it is interacting with a group of server objects, but rather “thinks’’ that it is interacting with an individual server object.
Repository Identifier	The identifier of a type within the Interface Repository.
REQUEST_DURATION_POLICY	A client-side policy that defines the time interval over which a client’s request to a server remains valid and should be retained by the server ORB to detect repeated requests.
FT_REQUEST Service Context	A service context, included in a request message, that allows a server to detect and suppress duplicate requests and to garbage collect requests that are obsolete.
Shared Logging	A logging strategy in which the primary member of an object group logs its state by writing the log records onto stable storage.
State Transfer	In both passive and active replication, when a new or recovered member of an object group is activated, a state transfer is required to transfer the state of the object to the new or recovered member, so that the new or recovered member will have the same state as the other members of the object group.

Stateless Object	The behavior of a stateless object is unaffected by its history of invocations. A typical example of a stateless object is a server that provides read-only access to a database.
Strong Membership Consistency	Strong Membership Consistency means that, for each method invocation on an object group, the Fault Tolerance Infrastructure on all hosts agree on the membership of the object group.
Strong Replica Consistency	For passive replication, Strong Replica Consistency means that, at the end of each state transfer, each of the members of the object group have the same state. For active replication, Strong Replica Consistency means that, at the end of each method invocation on the object group, each of the members of the object group have the same state.
TAG_FT_GROUP Component	A component of all of the profiles of the Object Group Reference that contains the fault tolerance domain identifier, object group identifier, and object group reference version number of the server object group with that reference.
TAG_FT_HEARTBEAT_ENABLED Component	A component of a TAG_INTERNET_IOP profile of an object group reference that indicates that a member of a server object group, or gateway, is heartbeat enabled.
TAG_FT_PRIMARY Component	A component of one of the TAG_INTERNET_IOP profiles of an object group reference that is intended to address the primary member of the object group, and that indicates that this TAG_INTERNET_IOP profile should be used in preference to other TAG_INTERNET_IOP profiles within the object group reference.
Total Order	The *ordered before* relation is the transitive closure of:
	• If message m1 is delivered to object replica O before message m2 is delivered to O, then m1 is ordered before m2. • If message m1 precedes message m2, then m1 is ordered before m2. • If both m1 and m2 are delivered to object replica O, and m1 is ordered before m2, then m1 is delivered to O before m2 is delivered to O.
	The ordered before relation is acyclic.
Unique Primary Replica	For passive replication, one and only one member of the object group executes the methods invoked on the object group.

Unreplicated Client Object	An unreplicated client object communicates with a replicated server object using IIOP. The client may communicate directly with a member of the server object group or, if multicasting is provided, the client may communicate with a gateway, which then multicasts the message to the server object group.
Virtual Synchrony	If object replicas O1 and O2 are in the same view of the object group membership M and they transition together to the next view of the object group membership M’, then the same messages are delivered to O1 and O2 while they are members of M. Virtual synchrony is used to ensure that a state transfer to initialize a new member of object group membership M occurs at the point in the message order corresponding to a membership change. Thus, at the start of the next view of the object group membership M’, all of the members in M’ will have the same state.
Warm Passive Replication	A form of passive replication in which only the primary member executes the methods invoked on the object group by the client objects. Several other members operate as backups. The backups do not execute the methods invoked on the object group; rather, the state of the primary is transferred to the backups periodically.

Appendix A Consolidated IDL

```
#ifndef _FT_IDL_
#define _FT_IDL_

#include “TimeBase.idl’’                 // 98-10.47.idl
#include “CosNaming.idl’’                // 98-10-19.idl
#include “CosEventComm.idl’’             // 98-10-06.idl
#include “CosNotification.idl’’          // from telecom/98-11-03.idl
#include “IOP.idl’’                      // from 98-03-01.idl
#include “GIOP.idl’’                     // from 98-03-01.idl
#include “ORB.idl’’                      // from 98-03-01.idl

#pragma prefix “omg.org”

module IOP {
    const ComponentId TAG_FT_GROUP = 27;
    const ComponentId TAG_FT_PRIMARY = 28;
    const ComponentId TAG_FT_HEARTBEAT_ENABLED = 29;

    const ServiceId FT_GROUP_VERSION = 12;
    const ServiceId FT_REQUEST = 13;
};

module FT {
    // Specification for Interoperable Object Group References
    typedef string FTDomainId;
    typedef unsigned long long ObjectGroupId;
    typedef unsigned long ObjectGroupRefVersion;

    struct TagFTGroupTaggedComponent { // tag = TAG_FT_GROUP;
            GIOP::Version               version;
            FTDomainId                  ft_domain_id;
            ObjectGroupId               object_group_id;
            ObjectGroupRefVersion       object_group_ref_version;
    };

    struct TagFTPrimaryTaggedComponent { // tag = TAG_FT_PRIMARY;
        boolean primary;
    };


    // Specification for Most Recent Object Group Reference
    struct FTGroupVersionServiceContext { //context_id = FT_GROUP_VERSION;
        ObjectGroupRefVersion   object_group_ref_version;
    };


    // Specification for Transparent Reinvocation
    const PolicyType REQUEST_DURATION_POLICY = 47;

    struct FTRequestServiceContext { // context_id = FT_REQUEST;
        string                  client_id;
        long                    retention_id;
        TimeBase::TimeT         expiration_time;
};

interface RequestDurationPolicy : Policy {
        readonly attribute TimeBase::TimeT request_duration_value;
};


// Specification for Transport Heartbeats
const PolicyType HEARTBEAT_POLICY = 48;
const PolicyType HEARTBEAT_ENABLED_POLICY =  49;

struct TagFTHeartbeatEnabledTaggedComponent {
// tag = TAG_FT_HEARTBEAT_ENABLED;
boolean heartbeat_enabled;
};

struct HeartbeatPolicyValue {
boolean                    heartbeat;
TimeBase::TimeT            heartbeat_interval;
TimeBase::TimeT            heartbeat_timeout;
};
interface HeartbeatPolicy : Policy {
        readonly attribute HeartbeatPolicyValue heartbeat_policy_value;
};

interface HeartbeatEnabledPolicy : Policy {
       readonly attribute boolean heartbeat_enabled_policy_value;
};


// Specification of Common Types and Exceptions for ReplicationManager
interface GenericFactory;
interface FaultNotifier;

typedef CORBA::RepositoryId TypeId;
typedef Object ObjectGroup;

typedef CosNaming::Name Name;
typedef any Value;
struct Property {
       Name nam;
       Value val;
};
typedef sequence<Property> Properties;

typedef Name Location;
typedef sequence<Location> Locations;
typedef Properties Criteria;
struct FactoryInfo {
       GenericFactory factory;
       Location the_location;
       Criteria the_criteria;
};
typedef sequence<FactoryInfo>  FactoryInfos;
typedef long ReplicationStyleValue;
const ReplicationStyleValue STATELESS = 0;
const ReplicationStyleValue COLD_PASSIVE = 1;
const ReplicationStyleValue WARM_PASSIVE = 2;
const ReplicationStyleValue  ACTIVE = 3;
const ReplicationStyleValue  ACTIVE_WITH_VOTING = 4;

typedef long MembershipStyleValue;
const MembershipStyleValue MEMB_APP_CTRL = 0;
const MembershipStyleValue MEMB_INF_CTRL = 1;

typedef long ConsistencyStyleValue;
const ConsistencyStyleValue CONS_APP_CTRL = 0;
const ConsistencyStyleValue CONS_INF_CTRL = 1;

typedef long FaultMonitoringStyleValue;
const FaultMonitoringStyleValue PULL = 0;
const FaultMonitoringStyleValue PUSH = 1;
const FaultMonitoringStyleValue NOT_MONITORED = 2;

typedef long FaultMonitoringGranularityValue;
const FaultMonitoringGranularityValue MEMB = 0;
const FaultMonitoringGranularityValue LOC = 1;
const FaultMonitoringGranularityValue LOC_AND_TYPE = 2;

typedef FactoryInfos FactoriesValue;

typedef unsigned short InitialNumberReplicasValue;
typedef unsigned short MinimumNumberReplicasValue;

struct FaultMonitoringIntervalAndTimeoutValue {
    TimeBase::TimeT   monitoring_interval;
    TimeBase::TimeT  timeout;
};

typedef TimeBase::TimeT CheckpointIntervalValue;

exception InterfaceNotFound {};
exception ObjectGroupNotFound {};
exception MemberNotFound {};
exception ObjectNotFound {};
exception MemberAlreadyPresent {};
exception BadReplicationStyle {};
exception ObjectNotCreated {};
exception ObjectNotAdded {};
exception PrimaryNotSet {};
exception UnsupportedProperty {
    Name nam;
    Value val;
};
exception InvalidProperty {
    Name nam;
    Value val;
};
exception NoFactory {
```

```
         Location the_location;
        TypeId type_id;
};
exception InvalidCriteria {
        Criteria invalid_criteria;
};
exception CannotMeetCriteria {
        Criteria unmet_criteria;
};


// Specification of PropertyManager Interface
// which ReplicationManager Inherits
interface PropertyManager {
        void set_default_properties(in Properties props)
                 raises (InvalidProperty,
                        UnsupportedProperty);

        Properties get_default_properties();

        void remove_default_properties(in Properties props)
                raises (InvalidProperty,
                        UnsupportedProperty);

        void set_type_properties(in TypeId type_id,
                                                in Properties overrides)
                raises (InvalidProperty,
                        UnsupportedProperty);

        Properties get_type_properties(in TypeId type_id);

        void remove_type_properties(in TypeId type_id,
                                                        in Properties props)
                raises (InvalidProperty,
                        UnsupportedProperty);

        void set_properties_dynamically(in ObjectGroup object_group,
                                                                in Properties overrides)
                raises(ObjectGroupNotFound,
                        InvalidProperty,
                        UnsupportedProperty);

        Properties get_properties(in ObjectGroup object_group)
                        raises(ObjectGroupNotFound);
};


// Specification of ObjectGroupManager Interface
// which ReplicationManager Inherits
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);

    ObjectGroup set_primary_member(in ObjectGroup object_group,
                                    in Location the_location)
            raises(ObjectGroupNotFound,
                   MemberNotFound,
                   PrimaryNotSet,
                   BadReplicationStyle);

    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);

};


// Specification of GenericFactory Interface
// which ReplicationManager Inherits and Application Objects Implement
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);
};


// Specification of ReplicationManager Interface
interface ReplicationManager : PropertyManager, ObjectGroupManager,
                               GenericFactory {
            void register_fault_notifier(in FaultNotifier fault_notifier);

        FaultNotifier get_fault_notifier()
                raises (InterfaceNotFound);
};


// Specifications for Fault Management
// Specification of PullMonitorable Interface
// which Application Objects Inherit
interface PullMonitorable {
        boolean is_alive();
};


// Specification of FaultNotifier Interface
interface FaultNotifier {
        typedef unsigned long long ConsumerId;

        void push_structured_fault(
            in CosNotification::StructuredEvent event);

        void push_sequence_fault(
            in CosNotification::EventBatch events);

        CosNotifyFilter::Filter create_subscription_filter (
            in string constraint_grammar)
                raises (CosNotifyFilter::InvalidGrammar);

        ConsumerId connect_structured_fault_consumer(
            in CosNotifyComm::StructuredPushConsumer push_consumer,
            in CosNotifyFilter::Filter filter) ;

        ConsumerId connect_sequence_fault_consumer(
            in CosNotifyComm::SequencePushConsumer push_consumer,
            in CosNotifyFilter::Filter filter);

        void disconnect_consumer( in ConsumerId connection)
                raises(CosEventComm::Disconnected);
};


// Specifications for Logging and Recovery
typedef sequence<octet> State;

exception NoStateAvailable {};
exception InvalidState {};
```

```
        exception NoUpdateAvailable {};
        exception InvalidUpdate {};


        // Specification of Checkpointable Interface
        // which Updateable and Application Objects Inherit
        interface Checkpointable {
                State  get_state()
                    raises(NoStateAvailable);

                void set_state(in State s)
                    raises(InvalidState);
        };

        // Specification of Updateable Interface
        // which Application Objects Inherit
        interface Updateable : Checkpointable {
                State  get_update()
                    raises(NoUpdateAvailable);

                void set_update(in State s)
                    raises(InvalidUpdate);
        };
    };
    #endif        // for #ifndef _FT_IDL_
```