FTS: A High-Performance CORBA Fault-Tolerance Service
A Technical Report

Erez Hadad

Computer Science Department
Technion - Israel Institute of Technology

Contents

1 Introduction 1
  1.1 Replication Model 1
    1.1.1 Object Group Replication 1
    1.1.2 Replica Consistency 1
    1.1.3 Fault-Detection 3

2 Top-Level View of FTS 4

3 FTS Components 5
  3.1 Object Replica 5
  3.2 Servant Factory 6
  3.3 Group Object Adaptor 6
    3.3.1 Outer POA 8
    3.3.2 Inner POA 8
    3.3.3 Outer Servant Locator (OSL) 10
    3.3.4 Replica Manager 10
    3.3.5 Invocation Logic 10
    3.3.6 Servant Registry 10
    3.3.7 Cache 10
    3.3.8 Multicast Request / Multicast Reply Servant 11
    3.3.9 Group Manager 11
    3.3.10 Quorum Client 11
  3.4 Group Communication Interface 11
  3.5 FTS Client Request Interceptor 12
### 3.5.1 Recommended Change to the CORBA standard

#### 3.6 FTS Server Request Interceptor

#### 3.7 Object Reference and the Fault-Tolerant Object Id

#### 3.7.1 Semantics

#### 3.8 Age and Primary View ID

#### 3.9 Lease Manager / Redirector (LMR) Service

### 4 Functional Description of FTS

#### 4.1 Client Redirection

#### 4.2 Request Processing

#### 4.2.1 Non-deterministic Request Execution

#### 4.3 Multicast Request Execution

#### 4.4 Request Caching and Completion

#### 4.5 Replica Management

#### 4.5.1 Sharing Multicast Messages Between Requests

#### 4.5.2 Service Acquisition

#### 4.5.3 State Transfer

#### 4.5.4 View Change

#### 4.6 Dynamic Object Management

#### 4.7 Dummy Reference Handling

#### 4.8 FTOID Implementation

### 5 Future Work on FTS

#### 5.1 Multi-Layering

#### 5.2 Multicast Reply

#### 5.3 Unimplemented Features and Immediate Extensions
1 Introduction

This document describes the detailed structure and operation of FTS, a CORBA fault-tolerance service designed for high-performance of replication and request processing. FTS is currently implemented in Java over Linux using ORBacus [3] as an ORB and Ensemble [1, 12] as a group communication toolkit. FTS research is described in [11]. A discussion of the qualities of FTS can be found in [9].

This document is organized as follows: next in this section is the outline of FTS’s model of operation. Section 2 provides a top-level view of FTS. Section 3 describes in high detail the components of the FTS infrastructure. Section 4 contains a functional description of FTS. Section 5 lists unimplemented features of FTS.

1.1 Replication Model

1.1.1 Object Group Replication

In FTS, replication is applied to disjunctive groups of server objects, rather than replicating each object using a separate group. Every server participating in a specific replication group holds replicas of all the objects that are replicated by the group, as described in Figure 1. By having replicas of different objects of the same service share group memberships, multicast operations and state transfer operations, the per-object GC resources, in terms of memory, CPU and messages, are reduced by a factor of the number of objects supported by the group. A further discussion of this is presented in [9].

Replicated object groups are heterogeneous in nature, i.e., objects of different types can share the same replication group. Also, a single FTS-based server process can support multiple replication groups of the service. On a broader scale, a single server machine can support multiple service applications. A server can join/leave replication groups dynamically or crash-fail. However, a replication group is assumed to never be left without members all at once: at least one server is supposed to survive until other servers are added to the group, thereby conserving the common group state. Note that guaranteeing this may require additional monitoring and management services, as described in [11].

FTS supports dynamic creation and deletion of replicated objects. This means that during state transfer, setting the new state includes not only setting the states of the existing object replicas in the server, but possibly also adding and deleting object replicas, automatically. In order to enable such changes in group replicas, FTS supports automatic dynamic creation and deletion of replicas.

1.1.2 Replica Consistency

FTS provides certain guarantees for the consistency between all the replicas of the same object, with respect to client requests. The following formal definitions for distributed consistency qualifiers are taken from [18, 7]:

Technion - Computer Science Department - Technical Report CS-2004-03 - 2004
Update Propagation: requires that each update to an object’s state is eventually received by all the replicas of that object.

Read Your Writes: requires that the effects of every update made by a client are visible to all subsequent queries of that client to the updated object.

Monotonic Reads: requires that the effects of every update seen by a client query are visible to all subsequent queries of this client (unless overwritten by later updates).

Monotonic Writes (FIFO): requires that all consequent updates requested by the same client are applied in the same order as they were issued.

Writes Follow Reads: requires that the updates whose effects are seen by a client query are applied before all subsequent updates requested by that client.

Total Ordering: requires that all the updates are applied in the same order in all the replicas.

By default, FTS guarantees update propagation, monotonic writes, writes follows reads and total ordering. This replication mode is called *loose consistency*. FTS can be also configured to use more strict replication modes, providing more guarantees but decreasing performance. *Sequential consistency* mode adds the guarantees of read your writes and monotonic reads for full sequential consistency [7]. In this mode, if a client accesses an
object sequentially through a single reference (which is the common case), then each of the client’s requests sees the effects of all of that client’s preceding updates, applied in their chronological order. Also, any client’s update to an object, once seen by a request, is consistently seen by following requests. Last, FTS enables linearizable consistency, similar to FT-CORBA’s strong replica consistency. This mode guarantees linearizability\(^1\) [5] of client requests. This is a requirement stronger than sequential consistency, implying that when an object is accessed in parallel through multiple references, every request sees the effects of all the clients’ updates that preceded it, as if those updates were applied sequentially to the object, but consistently with their global chronological order.

In all the replication modes, updates are totally-ordered using the GC multicast. In loose consistency mode, queries are executed immediately at the receiving server only. The resulting consistency operation is similar to the “Local Read” algorithm [5]. This mode is useful for services where performance is crucial and most of the operations are queries, such as distributed information services. In sequential consistency mode, execution of a client query is suspended until the target object becomes at least as up-to-date as it was in the end of the client’s most recent query/update. This considerably degrades query performance but enables a convenient programming model that is very popular in distributed systems. In some cases, such as clients holding multiple independent references of the same object, linearizable consistency is required. In this mode all client requests, queries and updates alike, are totally-ordered before execution. This mode yields the lowest performance but is also the least restrictive regarding the programming model.

The support for various replica consistency modes in FTS implicitly assumes that all the updates for the same replicated object are carried out in a uniform sequential order in all the replicas of the object. In order to handle multi-threaded clients and deferred execution in multi-threaded ORBs correctly, FTS provides two modes of parallel request processing. One is complete serialization of all the updates in a server, and the other is serialization of all the updates with the same target object. Obviously, full parallelism of request processing is sacrificed for the guarantee of a consistent order of execution. Moreover, FTS cannot control parallel processing of requests at non-CORBA objects, which could harm replica consistency. This is left as a responsibility of the application. A different approach is taken by Eternal [13], where full thread control is assumed by the fault-tolerance infrastructure. However, this is done by a proprietary interceptor of the threading mechanism at the OS level.

1.1.3 Fault-Detection

In FTS, two levels of fault-detection are offered:

**Process-based:** This is the basic level. In this level, monitoring is performed on the Group Object Adaptor (GOA) that manages a single replica of a group within a server process. As the GOA is essentially the membership of a server process in

\(^1\)The definition of strong replica consistency in FT-CORBA is not clear. Thus, we chose to follow its interpretation in Eternal [13] as linearizability of client requests.
a replication group, this boils down to relying on the group communication failure detection mechanism.

**Object-based:** This is the advanced level, in which in addition to the process based detection mechanism, the GOA on every node probes all objects registered with it by invoking their `is_alive()` method. Once failures are encountered, in the form of an invocation exception, the GOA automatically leaves the replication group and then re-joins it. Many of this method’s parameters are configurable, including, e.g., the time interval between probes, the number of objects to check each time, number of failures before taking action etc.

### 2 Top-Level View of FTS

A top-level view of the FTS infrastructure is presented in Figure 2. Each of the server application’s objects belongs to a *replication group*. All the objects in the same replication group are replicated together, i.e., their replicas are part of their group’s replicas. Each replica of the group is managed by a single Group Object Adaptor (GOA), which encapsulates all functions related to managing the replication of its registered objects and maintaining replica consistency. Each server can participate in multiple replication groups, by installing their respective GOAs and replicas. FTS imposes no restrictions on how to divide the objects of a specific application into replication groups. However, it is possible that a service administrator would add meaning to such a division, which would match actual deployment of the servers, e.g., subtrees of a naming service directory.

A client communicates through standard CORBA with a single object replica. Since FTS employs active replication, all replicas of all the objects within a group are active and may serve different clients at the same time in a parallel fashion. If the method invoked is defined as an *update*, i.e., may change the object’s state, then the GOA captures the request and sends it in a totally-ordered reliable multicast to the rest of the replicas using
the underlying group communication system. When receiving the multicast, the other replicas execute the request as well, thereby ensuring state consistency with the replica that was accessed by the client. Execution results of updates are cached, so that a client re-issuing the same request will receive a cached reply instead of causing re-execution of the request, thereby maintaining CORBA’s at-most-once execution semantics [15].

When a client’s communication with a server fails, the FTS Client Request Interceptor detects this failure, and locates, using the LMR service, an alternate server to be invoked. The interceptor then transparently redirects the request to the new server. The process may repeat, following consequent failures, until either the request completes successfully or a pre-configured timeout for client requests expires.

The GOA is integrated with the underlying GC system through a generic group communication interface (GCI), which enables the GOA to use most existing GC toolkits. Through the GCI, the GOA is aware of the replication group’s view. When the view changes, the GOAs in the group are notified, and, consequently, perform a state transfer, during which the replication group’s state, consisting of the replicated objects states and the cached replies, is passed to the new servers joining the group.

3 FTS Components

In this section we present a close look on the components of FTS, focusing on technical implementation aspects.

3.1 Object Replica

In order for an object to be replicated in FTS, it must be able to perform certain functions, which are defined in the interface StatefulObject, defined in Figure 3. The first and foremost of these functions is the ability to set and retrieve the object’s state, which is required for state transfer. This function is analogous to that of the Checkpointable interface in FT-CORBA [16].

Another function is that of the is_alive() method. This function must be implemented when object-level fault detection is employed. In this mode, the GOA periodically checks that all its registered objects are functioning correctly by invoking is_alive() on each object. The is_alive() implementation is expected to throw an exception in case of detecting errors, and may therefore include self-diagnostic code. This function is analogous to that of the PullMonitorable interface in FT-CORBA [16].

The last function required from a replicated object is derived from FTS’s support for dynamic destruction of replicated objects (Section 4.6). Consider the following scenario. Suppose that during state transfer, which follows a view change, a server that has just joined the group after being out-of-service for some time finds that some object replicas it supports no longer exist, e.g., because they were deleted while the server was disconnected. Following the state transfer, that server must implicitly remove such objects. Mere deactivation and/or deletion of the objects is not adequate since such actions might
interface StatefulObject {
    ObjectState get_object_state();
    void set_object_state(in ObjectState state);
    void notify_st_unregistration(in string goa_name, in ObjectId object_id);
    void is_alive();
};

Figure 3: StatefulObject Interface

3.2 Servant Factory

In order to support the dynamic creation of object replicas as described in Section 4.6, the GOA needs to be connected with machine-local factories that would generate object replicas of specified types, upon a request from the GOA. These factories must be able to generate only servants, i.e., unregistered objects, since the GOA itself handles the registration of the servants. The factories must implement the interface StatefulServantFactory, listed in Figure 4.

3.3 Group Object Adaptor

The GOA is the link between server application objects and the FTS infrastructure. Its functions include:

1. GOA is an object adaptor. Thus, once created, it is used for registering, (de)activating and managing information about CORBA objects, including extended reference comparison (Section 3.7.1). Also, as an object adaptor, it is responsible for injecting the FTOID component (Section 3.7) into the IORs of registered objects.
#define StatefulServant PortalbeServer::Servant
// In IDL, a Servant cannot be bound to an interface - this is
// just a reminder to the application programmer that the servant
// should implement the StatefulObject interface.

exception NoFactory {}; exception CannotCreate {
    string reason;
};

// StatefulServantFactory - (similar to the GenericFactory of the Life
// Cycle service) used for GOA-intenal creation of servants.
local interface StatefulServantFactory {
    StatefulServant create_servant(in CORBA::RepositoryId type_id,
        in GOA the_goa)
        raises (NoFactory, CannotCreate);
};

Figure 4: StatefulServantFactory Interface

Figure 5: The GOA Structure
2. GOA also serves as a high-level invocation interceptor. That is, it directs requests for which there is a cached reply to an internal reply cache instead of the actual object. Similarly, it directs other requests that must be multicast (updates) to go through the multicast based request execution mechanism.

3. GOA serves as a manager for a single replica in the replication group. Specifically, it participates in a replication group with other GOAs responsible for the other replicas of the corresponding objects. As a group member, GOA is responsible for handling synchronization during view change including state transfer (Section 4.5), for handling multicast messages and for participating in fault-detection, processor object-level, as described in Section 1.1.3. As a possible group coordinator, it is responsible for conducting the group initialization at view change and its registration as the primary partition for servicing clients.

4. GOA serves as a generic factory of objects. GOA needs to be associated with servant factories in order to support dynamic creation of replicated objects, as described in Section 4.6. The resulting ability to function as a generic factory is externalized and made part of the GOA interface so that applications can use it.

The GOA interface reflects the abilities described above regarding object management and dynamic object creation. It is listed in Figure 6.

Currently, there is no CORBA standard for building object adaptors in a portable fashion. However, the existing Portable Object Adaptor (POA) specification [15] is very flexible. POA can be used as a highly-customizable building block for more complex object adaptors, such as GOA. Figure 5 contains a schematic description of the structure of GOA. The major components of the GOA are listed in the following Sections.

### 3.3.1 Outer POA

The outer POA is used as a gate that controls the interaction of the server with clients. The requests arriving from clients must pass through this POA, since the published object references are registered with this POA. The outer POA is not much more than a gateway. It has no object table (NON_RETAIN policy) and uses a servant locator (USE_SERVANT_LOCATOR policy). During view change, the POA is closed for client requests by switching to DISCARDING state, and re-opens for service once the state transfer has completed by switching to ACTIVE state. Also, if the inner GCI communication becomes overloaded, the POA is temporarily switched to HOLDING state until GCI notifies that it can relay messages again. This POA is also used for inserting the FTOID component into the published IORs, as described in Section 4.8.

### 3.3.2 Inner POA

The inner POA is yet another gateway to the servants of the GOA, but one that is permanently open. It is considered a “back door”, i.e., the references of the servants that are
exception NoStatefulServant();
exception DuplicateRegistration();
exception ObjectNotRegistered();

local interface GOA {
    // GOA-specific API
    void get_tags(out string service_name, out string group_name);
    string name();

    // Object (un)registration API
    Object register_object(in StatefulServant _servant)
        raises (NoStatefulServant);
    Object register_object_with_id(in StatefulServant _servant, in ObjectId object_id)
        raises (NoStatefulServant, DuplicateRegistration);
    void unregister_object(in ObjectId object_id)
        raises (ObjectNotRegistered);

    // Object inspection API
    ObjectId reference_to_id(in Object obj)
        raises (ObjectNotRegistered);
    ObjectId id_to_reference(in ObjectId object_id)
        raises (ObjectNotRegistered);
    StatefulServant id_to_servant(in ObjectId object_id)
        raises (ObjectNotRegistered);
    ObjectId servant_to_id(in StatefulServant _servant)
        raises (ObjectNotRegistered);
    StatefulServant reference_to_servant(in Object obj)
        raises (ObjectNotRegistered);
    Object servant_to_reference(in StatefulServant _servant)
        raises (ObjectNotRegistered);

    // Object creation API
    void register_factory(in StatefulServantFactory _factory,
        in CORBA::RepositoryId type_id)
        raises (DuplicateRegistration);
    Object create_object(in CORBA::RepositoryId type_id)
        raises (NoFactory, CannotCreate, NoStatefulServant);
    Object create_object_with_id(in CORBA::RepositoryId type_id, in ObjectId object_id)
        raises (NoFactory, CannotCreate, NoStatefulServant, DuplicateRegistration);

    // Object comparison API
    boolean equivalent(in Object obj1, in Object obj2);

    // GOA control API
    void activate();
    void deactivate();
};

Figure 6: GOA Interface
registered with this POA are not published to the clients. Rather, they are used for provid-
ing privileged unblocked access for processing requests received by multicast from other
servers. This way, the multicast request can complete even if the group is closing client
service during view change and, as a result, multicast requests can complete execution
before state transfer.

### 3.3.3 Outer Servant Locator (OSL)

This servant locator is responsible for efficiently routing client requests received from the
outer POA to the appropriate mechanism for handling them. Upon receiving a client
request, the OSL first checks the cache to see if a reply is ready or is already being prepared
for this request, using the attached request id. If so, it directs the request for the cache to
handle. Otherwise, it consults with the invocation logic as to how to handle this request:
multicast the request then execute, execute then multicast the reply, or just execute the
request, depending on the request and the configuration, as described in Section 4.2. When
the behavior is decided, the request is accordingly routed to the multicast request servant,
multicast reply servant, or the target servant.

### 3.3.4 Replica Manager

Performs all the functions of replica management in the GOA, as described in Section 4.5.
The replica manager connects to the GCI as a client implementing GCI_Callbacks (Section
3.4), and processes events from the underlying GC system. It handles view changes, group
join/leave operations, unicast and multicast.

### 3.3.5 Invocation Logic

Decides which requests can be executed locally (queries) and which requests require mul-
ticast (updates). The decision is based upon what method is requested. More details of
its operation in Section 4.2.

### 3.3.6 Servant Registry

The servant registry is used for managing the servants that are registered with the GOA
as object replicas. In order to perform state transfer, all the registered objects must read
their state and add it to the global group state. However, efficient iteration over all the
registered objects is not available through the API of a standard POA. For that reason, this
special-purpose servant registry object is included in the GOA. Additionally, the registry
handles the age of the servants and that of the whole group.

### 3.3.7 Cache

Used for storing request/reply pairs for update-generating requests that have completed
processing, so that requests do not get executed more than once in each replica. The cache
also participates in the state transfer and is responsible for completing request execution. The cache operation is detailed in Section 4.4.

3.3.8 Multicast Request / Multicast Reply Servant

Generic dynamic invocation (DSI) servants used for multicasting requests before execution or multicasting requests+replies after execution.

3.3.9 Group Manager

The group manager is used when the GOA becomes the coordinator of a group. This object is used for processing messages received from all the group members during the group initialization phase. It also collects age and redirection information from the group members through the received messages. The messages are GCI unicast, relayed to the group manager by the replica manager.

3.3.10 Quorum Client

This object is used for handling the interaction with the LMR service. Using various access policies, it attempts to acquire a quorum of leases from a given pool of LMR servers. Then, it automatically maintains the acquired leases until it is told to abort.

3.4 Group Communication Interface

A group communication interface (GCI) connects the GOA and the GC toolkit, thus enabling the GOA the use of GC services of view management, unicast and reliable totally-ordered multicast. GCI relies on properties of GC that are common to most available toolkits: consistent view information including self membership and coordinator identity, view change events of begin view and view end, unicast between members, totally-ordered reliable multicast and primitives for joining and leaving a group.

GCI interface is event-based. Its main component is the GCI object (Figure 7), which is an abstraction of a group member. GOA, which acts as a client of GCI, implements an interface of GCI_Callbacks (Figure 8) - a listener for GC events, such as view change events or reception of messages, either by multicast or by unicast. The remaining components of GCI are abstractions of membership identity, membership list (Figure 9) and GCI message (Figure 10) that can be used for multicast or unicast. All GCI abstractions have back-end implementations matching the GC toolkit being used.

There are 4 states defined in GCI, in which a group member can be:

1. INIT: GCI is initialized but the member has not joined a group yet.
2. EXIT: GC system is terminating.
3. GROUPED: The member is in a group and can perform multicast and unicast.
4. VIEW_END: The current view has ended, waiting for new view to be established.

One may notice that the GCI contains no explicit support for state transfer. FTS performs state transfer and group synchronization following a view change using lower level primitives of multicast and unicast provided by GCI. This increases FTS portability, by not relying upon any specific model of state transfer. Also, it considerably simplifies the back-end implementation of GCI.

Currently, GCI is defined in Java. It has a back-end implementation for the Ensemble GC toolkit [12, 1]. However, as its model of operation is common to GC toolkits in general, we believe GCI can be ported to other toolkits just as well.

3.5 FTS Client Request Interceptor

The client request interceptor is attached to a client application’s underlying ORB. Its primary function is performing transparent client redirection at communication failures. That is, it detects faults in the communication between a client and a replicated object during an invocation and responds by redirecting the client request to a different replica of the object, as detailed in Section 4.1. Another task this interceptor is responsible for is obtaining a unique request id tag for every new outgoing request and attaching it to the request and its following retries, as a service context. This tag is used by the GOA’s caching mechanism to prevent duplicate execution of the request on the server side, similarly to FT-CORBA’s FT_REQUEST context. The client request interceptor detects non-FTS target objects and avoids interfering with their invocations.

The client request interceptor supports multiple parallel requests invoked using the same object reference, as is the case in multi-threaded clients or in deferred request execution. The problem in this case arises from the possibility that the target object, which is shared by multiple requests, might fail. Hence, multiple redirections might be performed for the same reference, causing not only an excess of messages, but also redirecting the reference to multiple different targets. In order to avoid this, a simple synchronization and versioning mechanism is employed that causes the first request that gets redirected to “lock” the reference until the redirection completes. The other requests are then forced to use the newly-obtained reference for invocation.

3.5.1 Recommended Change to the CORBA standard

The FTS client request interceptor generates a new unique request id tag and associates it with each new request issued by a client and with that request’s following retries. The only standard context in CORBA 2.3 [14] that uniquely associates every request with its following retries is PortableInterceptor::Current [17]. However, this context is writable only to client code, and is only indirectly available for reading its contents to the client request interceptor. Since FTS employs replication transparency, the client does not actively participate in the service and therefore cannot be asked to write data such as the request id to this context. Also, altering the standard by giving the client request interceptor write
public abstract class GCI {

    public static final int STATE_INIT = 0; // just initialized
    public static final int STATE_EXIT = 1; // terminating
    public static final int STATE_GROUPED = 2; // in group
    public static final int STATE_VIEW_END = 3; // current view ending

    // group state api
    public int GCI_state() {...}

    // group manipulation api - all methods are thread safe
    public abstract void join(String groupName)
        throws GCI_Exception;

    public abstract void leave()
        throws GCI_Exception;

    public abstract void new_view()
        throws GCI_Exception;

    public abstract void cast(GCI_Message msg)
        throws GCI_Exception;

    public abstract void send(GCI_Message msg)
        throws GCI_Exception;

    public abstract void terminate();

    // group information api (read only) - IS CORRECT ONLY WITHIN CALLBACKS!

    public abstract String group_name()
        throws GCI_Exception;

    public abstract GCI_Member[] view()
        throws GCI_Exception;

    public abstract GCI_Member self_member()
        throws GCI_Exception;

    public abstract GCI_Member coordinator()
        throws GCI_Exception;

    public abstract String view_id()
        throws GCI_Exception;

}

Figure 7: GCI Interface
public interface GCI_Callbacks {

    // New view is installed
    public void begin_view();

    // Current view is ending
    public void end_view();

    // A multicast message was received
    public void cast_received(GCI_Message msg);

    // A send (unicast) message was received
    public void send_received(GCI_Message msg);

    // GC subsystem is overloaded with messages
    public void overflow_set();

    // GC subsystem has overcome the message load
    public void overflow_clear();

    // Underlying GC is terminating
    public void exit();

}
public interface GCI_Member {

    // The host address of the member
    public String host();

    // The port of the GC protocol
    public String port();

    // The rank of the member in the group (not always valid)
    public int rank();

    // Is this member the elected coordinator?
    public boolean is_coordinator();

    // The full member’s identifier (optional)
    public String id();

    // Print method
    public String toString();

    // Comparison method
    public boolean equals(GCI_Member m);
}

Figure 9: GCI_Member Interface
public class GCI_Message extends ByteArrayOutputStream {

    // Constructor - specify type of message and to whom the message is
    // addressed to (can be NULL for multicast messages)
    public GCI_Message(byte msgType, GCI_Member toMember) {..}

    // Set the addressee (can be NULL for multicast message)
    public void to_member(GCI_Member toMember) {..}

    // Get the addressee
    public GCI_Member to_member() {..}

    // Get the sender
    public GCI_Member from_member() {..}

    // Get the message type
    public byte msg_type() {..}

    // Printing method
    public String toString() {..}

    // Return DataOutputStream to write to message
    public DataOutputStream new_DOS() {..}

    // Return DataInputStream to read from message
    public DataInputStream new_DIS() {..}

    // Return ByteArrayInputStream to read from message
    public ByteArrayInputStream new_BAIS() {..}

}

Figure 10: GCI_Message
access to the PortableInterceptor::Current breaks a general design concept of the Portable Interceptors specification [17] that does not permit a request interceptor to modify any request-associated data that is accessible to the client code.

In CORBA 2.4 [15] there is a new standard REQUEST context defined by FT-CORBA [16] that has qualities similar to those of the request id tag of FTS. This context can substitute for the request id resulting in an efficient and portable FTS implementation.

FTS does not, however, depend on the above REQUEST context. A transparent client side proxy [8] can be used to generate the request id and add it to the PortableInterceptor::Current. Yet, such a solution would decrease client side performance as a result of redirection of client requests back and forth between the client and the proxy. Thus, FTS currently employs a patch to the underlying ORB that would generate the request id tag in ORBs that do not support the above-mentioned REQUEST context. This patch works by injecting the request id into the client current just before the client interceptor is invoked. This patch does not use proprietary API, so it can be easily applied to many available ORB products.

3.6 FTS Server Request Interceptor

The server request interceptor is used in FTS as an extension of the GOA. As an object adaptor, the GOA has no standard access to service contexts, such as the request id, accompanying a client request. The server interceptor has access to this data and is used for passing this information to the GOA through a Request Context object [17]. It is also responsible for generating “dummy” request ids for requests from clients that do not participate in the service, thereby allowing the server to process those clients’ requests properly without harming replica consistency and/or integrity.

The server request interceptor also participates in the dummy reference handling optimization, described in Section 4.7. Similar to the client request interceptor, it detects requests aimed at non-FTS targets and avoids interfering with their processing.

3.7 Object Reference and the Fault-Tolerant Object Id

FTS utilizes the standard IOR without any structural or semantical changes. However, in order to support object replication, FTS manipulates IOR contents using facilities provided by the CORBA 2.3 standard [14], especially those enabled by the portable interceptors specification.

In FTS, a global name space is defined. This name space is independent of any physical deployment of FTS. The name space is divided, initially, into services, i.e., fault-tolerant service applications. Each service has its own unique service id. As each service’s objects are divided into groups, every object group within that service has its own service-scope-unique group id. The combination of the service id and the group id is also the assigned name of the Group Object Adaptor that manages a replica of the group at each participating server. Each object replica has its own CORBA-standard object id. All the object replicas which relate to the same object share the same object id. The service id, group
id and object id are combined together in a platform-independent format into a special component called the Fault-Tolerant Object Id (FTOID). This component is added as a CORBA tagged component to profiles in the IOR of each object’s replica, as described in Section 4.8.

When a failure in the communication with a server is detected, the FTS client-part uses the FTOID from the reference of the failed object in order to locate another replica of the same object and redirect the client’s request to it.

### 3.7.1 Semantics

FTS makes no change in the standard IOR structure. However, IOR semantics are slightly modified to accommodate for the existence of object replicas. Specifically, two (or more) CORBA object replicas are conceptually equivalent from the client point of view. In order to maintain client transparency, they should appear as such to clients, and in particular their corresponding IORs should be considered equivalent by IOR comparison utilities. At the server side, the application is aware of the replication. The GOA provides an explicit `equivalent()` predicate for an extended comparison of IORs, which considers group relevance by comparing the FTOID components of both IORs. Using this API, we made sure that IORs of replicas of the same object will always appear equivalent. At the client side things are more complicated. On one hand, recall that in FTS a client first gets an IOR that uniquely identifies the replicated object, and is then routed to the correct server, e.g., after failures, using `non-permanent` Location Forward exceptions. This keeps the IORs of the object before and after the failure equivalent. However, using the standard `is_equivalent()` method of the object interface, clients might detect two independently-obtained references of different replicas of the same object to be inequivalent. This issue can be handled using the reference comparison method of FT-CORBA [16], which extends the `is_equivalent()` method to compare IOGRs based on their `TAG_FT_GROUP` component. By having FTS generate IORs that include this IOGR component, we can achieve full replica equivalence at the client side as well.

### 3.8 Age and Primary View ID

In FTS, when members of a replication group install a new view, they also agree on the common state of service by transferring the state of one of the members, designated as `provider`, to the others. Selecting a provider that is less up-to-date than others would lead to a loss of client updates. To avoid this as much as possible, FTS attaches an `age` to each of the members, and one of the "oldest" members (members with highest age) is the designated provider.

In order to define a meaningful age to a server, which closely reflects how up-to-date is that server, FTS uses the following construction:

1. Each view is labeled with a unique `view id`, known to its members, and generated by the underlying GCI mechanism.
2. A server can receive updates only when it is in a view of a primary partition (Section 4.5.2). Each primary partition’s view is labeled with a (Primary View ID (PVID) that consists of the view id and a Primary View Counter (PVC). From the service acquisition algorithm described in Section 4.5.2, primary partition views can be installed only one after the other has ended. Also, each new PVID has a PVC that is higher than its predecessor, due to the LMR service (Section 3.9).

3. An age consists of a Primary View ID (PVID), a Consequent View ID (CVID) and an Update Count (UC). The CVID is the view id of the view installed by server after the primary view with the PVID has ended. The UC is the total number of updates processed by the server.

The GC guarantees that every two servers receive the same set of updates (in the same order) that were multicast during one view if those servers install that view and a succeeding common view. Combined with the state transfer in the beginning of a view, we can construct an age the consists of two view ids of two consequent views (PVID and CVID). If two servers have the same age, they are bound to have the same state. The PVC part of the PVID is used to tell whether one server is more up-to-date than another. Last, the UC is used when two servers are of different age but equally "old" (same PVC). In this case, FTS would select the server with the higher UC as the provider, as it processed more updates that its peers.

The age concept is also applied separately to each object inside a replica. This age is used to improve the efficiency of the state transfer, as described in Section 4.5.3. Since an object age needs two consequent view ids, the following technique is used: every time an object gets updated, it is marked as dirty and has the PVID set in its age. When the next view is installed, the servant registry scans all the dirty objects and sets the CVID in each object, thus clearing the dirty flag.

3.9 Lease Manager / Redirector (LMR) Service

LMR servers performs two major functions. One function is to provide leases for groups that try to become primary partition (Section 4.5.2). The other function is to redirect failed client invocations to alternate servers in the save service and group. The interface of an LMR server is described in Figures 11, 12 and 13.

Each LMR server maintains a set of leases. Each lease is for s different context, which is a label constructed from the service and group labels of each replication group. As a result, multiple partitions of the same replication group contend for the same context. Additional information is stored for each lease: minimal PVID (Section 3.8) and redirection information. Any partition requesting a lease for a specific context needs too have a server who has installed a primary view whose id is not lower than the minimal PVID associated with that context. After a new primary partition is established, its new PVID is committed to the LMRs in its quorum as the new minimal PVID (monotonically growing). As a result, new primary partitions are always at least as "up-to-date" as their predecessors. The
redirection information is updated together with the minimal PVID when a new primary partition is established, by invoking the set_entry() method of the LMR. When a client invocation fails, this information is used to redirect the client to a new server, as described in Section 4.1.

4 Functional Description of FTS

In this section we provide a dynamic view of FTS, by describing key processes in the system that ensure its operation.

4.1 Client Redirection

Fault-tolerance is virtualized in FTS through the use of client redirection. It is available only for clients participating in the service, i.e., clients whose ORB function is augmented by the FTS client request interceptor.

The FTS client request interceptor receives indications of errors in the client’s interaction with the server during request processing in the form of CORBA exceptions, prior to alerting the client code. The interceptor reacts only to those exceptions that match “fail-over conditions”, as described in the FT CORBA specification [16].

In order to redirect the request to an alternate server, the interceptor selects an LMR server from a list of predefined locations and invokes the redirect() method of the LMR server with the IOR of the failed target as a parameter. The LMR extracts the FTOID component from the failed IOR and replies with an alternate server selected from the redirection information associated with the matching context.

If the LMR has no matching context it responds with a NoMatchingContext exception, and as a result the client interceptor switches to another LMR. If the LMR
// A shorter name
typedef FTS_ViewId::PrimaryViewId PrimaryViewId;

// Definition of key type
typedef long LeaseKey;

// Definition of lease time
typedef long LeaseTime;

// lcontext is already locked by another valid key
exception LockedContext {};

// The lease of the given key has expired
exception LeaseExpired {};

// The PVID in the request is lower than the PVID in the lease manager
exception LowPVID {};

// Defines the mechanism of locking
interface LeaseManager {

    // Obtain a new lock for a given lcontext for a given period of
    // lease time (msec). If the lcontext is already leased,
    // LockedContext is thrown. If the PVID is lower than
    // the LM’s, LowPVID is thrown.
    LeaseKey lock(in string lcontext, in LeaseTime ltime, in PrimaryViewId pvid)
        raises (LockedContext, LowPVID);

    // Update a lease on a lock using a given key, and create a new
    // key. If the lcontext is leased by another key, LockedContext
    // is thrown. If the lcontext is not leased, the operation becomes
    // lock(). If the lease key is still valid but the PVID is lower
    // than the LM’s, LowPVID is thrown.
    LeaseKey relock(in string lcontext, in LeaseKey lkey, in LeaseTime ltime,
        in PrimaryViewId pvid)
        raises (LockedContext, LowPVID);

    // Release a lock on a lcontext. If the lcontext is locked by a
    // key other than the given key, LockedContext is thrown.
    void unlock(in string lcontext, in LeaseKey lkey)
        raises (LockedContext);
}

Figure 12: Lease Manager Interface

exception NoAlternateServers {};

exception NoMatchingContext {};

interface Redirector {
    Object redirect(in Object failed_object)
        raises (NoAlternateServers, NoMatchingContext);
};

Figure 13: Redirector Interface
has redirection information, it either replies with an alternate server’s dummy IOR or a
NoAlternateServers exception if there aren’t any. In the case of no direction information,
the client simply attempts another LMR from a predefined list. Once a primary partition
of the service group is set, the client is bound to find an LMR with proper redirection
information.

Once having obtained an alternate target reference, the interceptor redirects the client
request to the new target by throwing a non-permanent Location Forward exception\(^2\),
thereby causing the ORB to redirect the request to the new target transparently to the
client.

If a client invocation is redirected by an LMR server that does not have the latest
redirection information for that specific replication group, either the client would reach a
server that is operational and continue with its work, or the client invocation would fail
again and the process will repeat until success or a predefined timeout.

### 4.2 Request Processing

FTS distinguishes between client updates and queries. Updates are requests that might
cause a change in a replicated object’s state, whereas queries just read the state without
changing it. The distinction is performed at the level of the replicated object’s IDL in-
terface, by tagging the method identifiers using an external configuration file. This also
enables modifying the tagging without recompiling the server code.

In the default loose consistency mode, the Local Read algorithm \([5]\) is applied: Client
queries are executed locally and return immediately upon completion. Client updates first
undergo totally-ordered reliable multicast and return after having been received back from
the multicast and executed. All the requests received by multicast are executed upon
reception. In the linearizable consistency mode, all client requests are considered updates,
undergoing multicast and execution upon reception. The sequential consistency mode is a
future design extension, detailed in Section 5.3.

Multicasting CORBA requests requires a dynamic mechanism for serializing the re-
quests into multicast messages and then de-serializing and executing the requests upon
reception. The standard CORBA request structure has no API support for serialization.
Therefore, FTS implements, using only standard CORBA components, its own serializable
request container objects. These objects are capable of storing all the request and reply
contents including target object id, method id, parameters (in/out/inout), result, excep-
tions and the request id. A request container retrieves its request data from a standard
DSI ServerRequest invocation structure. After de-serialization, the request container can
be connected to the receiver GOA’s inner POA, and perform a standard dynamic invoca-
tion (DII) of the request using its stored object id. The dynamic invocation can be either
synchronous or deferred.

Reading DSI ServerRequest information requires, as a parameter, the type information

\(^2\)In CORBA 2.4 there is only one kind of LOCATION FORWARD exception, which is equivalent to
the non-permanent LOCATION FORWARD in CORBA 2.3.
of all the request components, i.e., parameters, exceptions and result. This information is obtained from the interface repository [15]. However, reading type information from the interface repository for every request is wasteful, since this is a slow operation, and the obtained type information does not change during runtime. FTS accelerates the process by accessing the interface repository only during server startup and reading the type information of all the methods. This information is stored in empty request containers, which are grouped together in the invocation logic - a dictionary-like structure whose keys are the interface and method identifiers. Upon each method call, the proper empty request container is obtained from the invocation logic and cloned into a new one. The clone is then used for the actual DSI information retrieval and for the rest of the process.

When a server (more specifically, a GOA) initializes, the invocation logic reads the external configuration file that contains the method tags and attaches the tags to the request containers in the dictionary. Upon a request invocation, the outer servant locator (OSL) reads the interface and method ids attached to the request and passes them to the invocation logic. If the request requires multicast, the invocation logic obtains the matching tagged request container and returns it to the OSL (otherwise NULL is returned). The OSL uses the returned information to direct the request to the proper execution mechanism. In case of a query, i.e., a request that requires no multicast, the request is directed to the target servant for execution. In case of a multicast, the request container is cloned and the request is directed to the multicast request servant. This servant (Section 3.3.8) is a DSI servant that reads the invocation information and stores it into the request container. Then, the request container is sent to the replica manager for multicast. Upon reception of a request container by multicast, the replica manager connects the request container to the inner POA and executes the request dynamically on the target servant. If the request was originated by the receiving server, the reply is returned to the client. The flow of the operation is detailed in Figure 14.

4.2.1 Non-deterministic Request Execution

By using the method tagging mechanism, FTS provides support for non-deterministic requests, such as requests involving random numbers generation, time-dependent computation or sensor sampling. If such a request is a query, it can be tagged and safely handled as a non-multicast request. In case of an update, the non-deterministic nature of the request might break the replica consistency when the request is executed in each of the servers. In this case, the problem can be handled by separating the non-deterministic part of the computation and defining it as a separate query, which is called from inside the original update method. However, this gives rise to another problem of multi-layering, which is discussed in Section 5.1.

4.3 Multicast Request Execution

When a request is delivered from the multicast mechanism, it needs to be executed by the receiving server. If the request is executed synchronously by a single receiving thread,
Multicast Request Servant Invocation Logic

Outer Servant Locator

Outer POA

Client Invocation

1. Client Invocation

2. Select RC [no cached reply]

3. Execute Request in Servant (*)

4. Put RC of request in cache

5. Pass RC with DSI parameters

6. Clone RC with DSI parameters

7. Pass RC

8. Retrieve reply from cache (*)

9. Multicast RC

10. Multicast Delivery

11. Complete deferred invocation and DSI invocation (*)

12. Execute Request

13. Execute Request [cached reply]

14. Retriever reply [cached reply]

15. Complete deferred invocation

16. Complete deferred invocation

Notes:

1. Square brackets [] define a condition that enables the operation.

2. (*) marks the end of request processing. If the condition is met, the request is immediately returned to the client.

3. The dotted lines mark automatic POA operations.

4. The request reply may be forwarded to the client if no cached reply is found.

5. The dotted lines mark automatic POA operations upon invocation.
then all the requests are executed one after the other. Depending on the threading model (Section 1.1.2), this may be useful, but in many cases it degrades performance considerably. Moreover, in some GC toolkits, such as Ensemble [1, 12], there is only one thread for dealing with all the GC events. Therefore, keeping that thread busy executing requests might significantly delay the server’s responsiveness to other important events such as view change.

Customarily, a threaded execution engine would increase parallelism. Such an engine would probably have a limited thread pool in order to avoid thread proliferation, but would still be wasteful, since all of its threads would be client threads, whose sole purpose is to wait for the request execution completion. FTS employs a more efficient solution by utilizing *deferred synchronous* request execution [15]. This way, the thread receiving the message signals a single control thread of the new task. The control thread, in turn, dispatches requests for execution in deferred mode, never waiting for the requests to complete. When a request is done executing, it begins its grace count until it is removed from the cache. Also, on the server that issued the multicast on behalf of the client, the multicast servant is notified. Then, it completes the request processing and returns a reply to the client.

The flow of processing a request received by multicast is described in Figure 15.

### 4.4 Request Caching and Completion

Following CORBA’s at-most-once semantics [15], a client request that modifies the state of a replicated object must not be executed more than once in each of the object’s replicas, or an inconsistency would appear between the actual state of the replicated object and its request execution history. For that reason, such requests, once finished processing, are stored, along with their matching replies, in the GOA’s cache, inside request containers. Should the client retry to invoke the same request at a server following redirection, and had that request already been processed in the target server, the cached reply would be returned instead of re-executing the request. The cache is implemented as a DSI servant so that it can handle requests dynamically and independently of any particular interface.

Matching a received request with the cached request/reply is done by comparing the request id tag attached to the request with those of the cached request containers. This tag uniquely identifies a specific request and is attached to the original request and its following retries by the client request interceptor, as described in Section 3.5.

Inherently, the cache needs to handle a possible race condition between a thread handling a client request and the control thread that handles the execution of requests received from the multicast stream. In one possible case, a client may re-issue a request to a server that has already received a multicast of the client’s original request, but has not yet finished processing it, so the reply does not appear in the cache yet, and thus the request might be executed twice. A second case is when a client re-issues a request to a server before that server receives the multicast of the original client’s request, possibly leading again to double execution of the client’s request. This race is solved by having both threads try to put the request in the cache prior to proceeding with execution. The first to do it gets to execute it. As for the "loser" thread, its following action depends on its identity: the
1. Square brackets [ ] define a condition that enables the operation.
2. (*) marks the end of the request processing. If this is completed, the request may be returned to the client.
3. The dotted lines mark automatic POA operation upon invocation.
4. Complete deferred invocation (*)
multicast thread will simply abort processing that request; the thread of the client request will wait until the request is done processing, retrieve the reply and deliver it to the client. The unique request id attached to the request, independently of how it arrives, ensures that a request is executed at once in the same server. Note, however, that this does not apply to requests originating from clients that do not participate in FTS, as such requests do not have associated unique ids.

The cache must be purged from old requests/replies, so that it does not grow indefinitely. For that purpose, the cache employs a purging thread that performs a purging sweep once every configured time interval. Each entry in the cache that has finished execution, i.e., is not “pending” anymore, is assigned a positive grace count that is decreased by one in every sweep. Once it reaches zero, the entry is removed. The grace count is high enough so that it would expire after a client timeout, otherwise a request might still be executed more than once.

As noted previously, multicast requests are executed in deferred synchronous mode, without waiting for their completion. Still, the cache must be notified of the requests’ completion, so that it can later safely purge their entries. This is achieved by having the cache’s purging thread check the pending requests for completion and retrieve the results in case of completion. This mechanism is carefully synchronized with client invocation, which would try to get the results separately. The operation of the cache during request processing is described in Figures 14,15.

The cache contents are part of the group replica’s state, along with the states of all the replicated objects. This is because servers joining the group may later receive re-invocations regarding requests that may have already completed execution, and must be able to reply with only cached results so that at-most-once semantics are not violated.

Another role of the cache during state transfer, beside transferring its contents between servers, is to notify when a server is ready to transfer its state. When the current view of the replication group ends, there may still be requests being executed. There is no point in reading the state of an object on which requests are being executed at the time of the read, since there is no clear notion of which requests have completed before the read state and which requests still need to be executed. The object’s state is meaningful only when no requests are being executed. We say that an object becomes coherent with its execution history when it is not executing any requests. At the group level, the cache is aware of whether requests are executing or not, and is therefore responsible for notifying when a group replica reaches coherence as a whole.

4.5 Replica Management

As a member of a replication group, using GC subsystem, the GOA manages various GC events. View change events are used for triggering a service acquisition algorithm for selecting a primary partition, as well as for transferring the group state, i.e., the replicated objects’ states and the cache contents, from veteran servers to new servers joining the group. Message reception events are used for distributing updates as well as for synchronization. The GOA also initiates operations of joining/leaving groups and efficiently batching update
multicasts to the group. This functionality resides within the replica manager component of the GOA (Section 3.3.4).

### 4.5.1 Sharing Multicast Messages Between Requests

In FTS some of the client requests undergo a totally-ordered reliable multicast. Such multicast is expensive in terms of messages and computation resources. However, the per-request overhead can be reduced by having multiple requests share a single multicast operation by grouping the requests into a single multicast packet, as described in [10].

When a request is sent to the replica manager for multicast, it is queued. Once every configurable time interval, or when necessary, such as during view change, a multicast thread gathers all the queued requests into a single message which is then multicast using the underlying GCI object (Section 3.4). Depending on the application and the deployment of the servers, the operation interval of the multicast thread must be carefully selected. An interval too large would increase client waiting time, while an interval too small would cause small multicast packets and, as a result, low multicast throughput.

### 4.5.2 Service Acquisition

In a single well-connected network without communication interruptions, there should be only one server group replicating each group of objects. However, network partitions and communication disruptions might cause a replication group to split into several sub-groups, or partitions. If each partition is permitted to continue serving clients independently of the other partitions, their group states might become inconsistent with one another, making it very hard to merge again after the network has healed. FTS avoids this by permitting only one partition, designated as the primary partition, to continue serving clients, while having the other partitions suspend service until connectivity is restored and they can merge with the primary partition group.

Primary partition selection is through a technique called service acquisition, based on the quorum resource principle [6]. Each of the partitions belonging to the same original service group tries to acquire the ownership of the service provided by the original group (i.e., become its sole active representative), by acquiring leases for operation from LMR servers (Section 3.9). Partitions that cannot access any LMR servers forfeit automatically. The ones that do, still need to acquire leases from a quorum of LMR servers. Since a lease can be given only to one requester at a time and all quorums intersect (i.e., the conjunction of every two quorums is not empty), at most one partition can acquire a quorum of leases, and thus get to serve clients. The other contestants withdraw into a time-limited sleep, after which they shall retry to take ownership of the service. The service acquisition algorithm gives preference to continuous ownership by making a primary partition continuously renew its leases as long as it has access to the right LMR servers. However, once leases are not renewed due to server failures or network split-ups, they expire and another partition may acquire them.

Another property maintained by the service acquisition algorithm is that every new
primary partition should be at least as up to date (regarding its state) as its predecessor. This is implemented by having a *minimal PVID* (Section 3.8) label associated with each lease. When a partition requests a lease, it has to report a PVID that is not lower than the minimal PVID associated with that lease in the LMR server. When a new primary partition is formed, it creates a new PVID that is higher than any of those stored in its quorum of LMR servers, and updates the LMR servers, thus increasing their relevant minimal PVIDs.

An underlying assumption of FTS is that the GC system automatically detects changes in reachability of servers and signals installation of matching views. As a result, a primary partition server might migrate to another partition, transfer its state to its servers (Section 4.5.3) and make it a candidate for a new primary partition.

### 4.5.3 State Transfer

Each time a replication partition is set, new servers may have joined the partition. The term “new” does not necessarily indicate servers that were recently instantiated, but servers that were unable to participate in the previous primary partition. Those servers were not, therefore, able to serve client requests, and as a result, their replicas were not updated. In order to become as up-to-date as the “veteran” servers, the new servers receive a record of the state of the entire group replica. This state consists of the updated states of object replicas, indications of object replicas that were dynamically added to or removed from the group, cache contents and PVID of the server (which may be higher than the highest PVID of an object update due to primary partitions serving with no updates).

In FTS, one of the “veteran” servers in the new view is designated as *provider* of the state. The provider’s election is the coordinator’s decision based on the age data sent to it by all the members. The provider prepares a state structure and multicasts it to the group. The new joiners, or the *consumers*, use the received data to update their state.

A state update in a server consists of several operations. First, it compares its objects contents and age with those in the state record. When an object’s age differs from the record, the object’s state is reset to match the record. Objects that appear in the record but not in the server’s data are instantiated with the stored state, using the GOA’s generic factory (Section 4.6). Objects in the server’s data that are missing from the record are notified to self-destruct (Section 3.1). The objects are updated this way instead of a complete rewrite to save time when the difference between the record and the replica are small. Next, the cache is completely rewritten by the recorded state. Last, the server’s PVID is set to the recorded PVID.

### 4.5.4 View Change

This section presents a summary of the view change sequence applied at every replication group. An overview of the sequence appears in Table 1. Note that the coordinator GOA also performs as a member GOA.
<table>
<thead>
<tr>
<th>Event</th>
<th>Member GOA</th>
<th>Coordinator GOA</th>
</tr>
</thead>
<tbody>
<tr>
<td>View End</td>
<td>1. Suspend client service by setting outer POA to DISCARD-ING state.</td>
<td></td>
</tr>
<tr>
<td>New View</td>
<td>1. Send member data (server age, dummy reference) to coordinator</td>
<td></td>
</tr>
<tr>
<td>Received member data</td>
<td>1. Find PVID with maximal PVC from ages of members.</td>
<td>1. Find PVID with maximal PVC from ages of members.</td>
</tr>
<tr>
<td>from all members</td>
<td>2. Use the maximal PVID to obtain leases from a quorum of LMR servers.</td>
<td>2. Use the maximal PVID to obtain leases from a quorum of LMR servers.</td>
</tr>
<tr>
<td></td>
<td>3. If successful, skip to step 5.</td>
<td>3. If successful, skip to step 5.</td>
</tr>
<tr>
<td></td>
<td>5. Compute new PVID as (Maximal PVID with PVC++)</td>
<td>5. Compute new PVID as (Maximal PVID with PVC++)</td>
</tr>
<tr>
<td></td>
<td>6. If member ages are different, decide to do state transfer, otherwise,</td>
<td>6. If member ages are different, decide to do state transfer, otherwise, decide</td>
</tr>
<tr>
<td></td>
<td>decide not to.</td>
<td>not to.</td>
</tr>
<tr>
<td></td>
<td>7. If decided to do state transfer, select the member with the highest</td>
<td>7. If decided to do state transfer, select the member with the highest age as</td>
</tr>
<tr>
<td></td>
<td>age as provider of state.</td>
<td>provider of state.</td>
</tr>
<tr>
<td></td>
<td>8. Cast IN, with new PVID, decision of state transfer and age and id of</td>
<td>8. Cast IN, with new PVID, decision of state transfer and age and id of provider.</td>
</tr>
<tr>
<td></td>
<td>provider.</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: View Change (continues on the next page)
<table>
<thead>
<tr>
<th>Event</th>
<th>Member GOA</th>
<th>Coordinator GOA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Received OUT</td>
<td>1. Do not expect any further messages until view is changed</td>
<td>1. Stop refreshing any held leases</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Set timer to force view change after a predefined timeout</td>
</tr>
<tr>
<td>Received IN</td>
<td>1. If not doing state transfer, set new PVID and send \textit{ack(new PVID)} to coordinator.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. If doing state transfer:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(a) Select role: if id==self, then \texttt{provider}. Else, if age != (age of provider) then \texttt{consumer}, else \texttt{none}.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(b) If \texttt{provider}, prepare \textit{state} message and cast it.</td>
<td></td>
</tr>
<tr>
<td>Received \textit{state}</td>
<td>1. If \texttt{consumer}, set state.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. Set new PVID.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. Send \textit{ack(new PVID)} to coordinator.</td>
<td></td>
</tr>
<tr>
<td>Received \textit{ack(new PVID)} from all members</td>
<td>1. Update LMRs with redirection information and new PVID</td>
<td>2. Cast \textit{GO}.</td>
</tr>
</tbody>
</table>

Table 1: View Change (continues on the next page)
### 4.6 Dynamic Object Management

FTS supports dynamic creation and deletion of replicated objects. Currently, a created object is added to the same group of the creating object. FTS offers two mechanisms of object creation: one is by creating a servant and registering it at the GOA, and the other is by requesting the GOA to create a new object of the requested type. In order for the GOA to support dynamic creation of objects of certain types, the GOA is connected, by server local registration, with factories that can generate object replicas of the required types, upon a request from the GOA. These factories (Section 3.2) must be able to generate only servants, i.e., pre-registered objects, since the GOA itself handles the registration. The GOA’s ability to dynamically create objects is especially required during state-transfers, when a server may be required to implicitly generate new objects in order to match its group-state with that of other servers. During state transfer, implicit dynamic object deletion may also be required. It is enabled through the interface of the object replica, as detailed in Section 3.1.

### 4.7 Dummy Reference Handling

In order to provide an alternate object reference during client redirection, the group manager must have information of all the available object replicas in the group. This information is passed by unicast from each member to the coordinator during view change synchronization (Section 4.5.4). A single CORBA reference size is quite large (0.5KByte or more). This could hamper group scalability from the perspective of number of objects replicated per group, by significantly slowing view change synchronization for groups supporting a large amount of replicated objects. As a solution, each member sends to the coordinator only one dummy reference, pointing to that member’s GOA and ORB, but not to any specific object in the GOA. During client redirection, the LMR server extracts the server information from the failed reference and returns the dummy reference of an alternate server, which is then returned to the client interceptor. The client interceptor throws a non-permanent Location Forward exception [15], thus redirecting to the dummy reference but keeping all the original IOR components including the FTOID (Section 3.7). When the client invokes the dummy reference, the outer servant locator at the alternate server detects the dummy reference. Using the FTOID information, it directs the invocation to the correct servant, and returns the correct object reference to the client inside a service...
context, in the end of the invocation. The client interceptor detects the returned corrected reference in the service context, and in the next invocation to that object, it throws another non-permanent Location Forward exception, locally redirecting the invocation to the actual object.

4.8 FTOID Implementation

The FTOID component (Section 3.7) is initially inserted into the IOR of an object using the IOR interceptor mechanism introduced by the portable interceptors specification [17]. The component is provided as a policy object assigned to the outer POA, on which the published IORs are generated. When an IOR is generated in the outer POA, the policy containing the component is passed to the IOR interceptor, which converts it into a component and inserts it into the generated profile of the IOR. The component initially contains a blank object id field, since it is inserted to all the generated IORs of the same POA. Later, the object id is modified to match every specific object by manipulating the contents of the IOR using standard IOR structure definitions [15].

FTOID data needs also to be accessed by the client request interceptor. This is provided in a straightforward fashion by the standard interface of the client request interceptor [17], which enables direct access to IOR profiles and components.

5 Future Work on FTS

In this section we describe features of FTS whose implementations are expected to be completed in the near future.

5.1 Multi-Layering

A fundamental principle in CORBA service implementation, and in component software in general, is that of layering: several basic service objects are combined in order to form a more complex service object implementation. As a result, invoking a method in a service object may cause that object to generate subsequent method invocations on other service objects, as their client.

Note that unless handled properly, multi-layering may cause serious performance problems and even incorrect behavior. This is because if a method executed by each replica of an object invokes a method on a second replicated object, the subsequent invocation will be issued multiple times, which may violate the at-most-once semantics. If the secondary object is also replicated by FTS, all replicas will produce the same request id, and thus the redundant requests will be recognized as duplicates, and suppressed. In this case correctness will be preserved, but still too many messages will be generated. This problem still needs exploring.
5.2 Multicast Reply

If a request that requires heavy computation is tagged as an update, then all the replicas process it. As a result, all the servers become loaded and total client service performance decreases. Merely separating the extreme computation parts into a separate query, similar to the non-deterministic requests, is insufficient, since when processing a query, a connection disruption between the client and server might force the client to direct its request to a different server, while the original server may still be processing the request. This may cause yet another proliferation of the computation. Another case where we wish to reduce the computation to a single server is one-shot queries, i.e., reading data that is available only once, such as some types of data acquisition channels. For this type of problems we plan to extend FTS to include multicast reply. Tagging a method as multicast reply means that it is executed by the server accepting the request, and only afterwards the request+reply are multicast to the other servers to be cached there. This mode is safe only for queries.

5.3 Unimplemented Features and Immediate Extensions

FTS development is an ongoing process. Some of the features of FTS that are yet to be fully implemented include:

1. **Object-Level Fault Detection**: This involves the GOA being able to incrementally check all of its supported objects, at a constant rate. Currently, FTS implementation supports only process-level fault-detection.

2. **Shared Multicast**: Currently, FTS supports only multicast-per-request, where each update request is multicast, once received from the client, to the rest of the replicas. Better performance can be achieved by grouping update requests into batches and multicasting them together.

3. **Update Execution Concurrency**: Currently, updates are distributed to all the servers in a totally-ordered fashion, but are then executed in each server in parallel, harming the consistency. The mechanism that executes received updates in each server needs to be extended into enforcing certain orders of execution: One that simply executes updates in the order they are received and one that parallelizes execution of updates that target different objects (but keeps the total order between updates for the same object).

4. **Consistency Modes**: Currently, FTS supports only loose consistency. Linearizable consistency is a simple extension of totally-ordering all the requests. In order to support sequential consistency, the following mechanism is considered. Every object has an age tag attached to it, counting the updates that were applied to the object. Every client query is accompanied with a tag specifying the age of the target object as seen by the client in its previous request (query/update). A query execution is
suspended until the target object “reaches” the specified age by applying updates. Since the object is a compound data type, the object’s age is meaningful only when no updates are being performed on the object, so only then should queries be enabled. Therefore, by using some synchronization/scheduling mechanism, queries need to be "inserted" into the update processing stream and computed.

5. **Negotiating multiple LMRs**: The current implementation of FTS works with a single LMR server. Negotiating multiple LMRs first requires implementing a "quorum client" engine that can accept various policies for accessing LMRs, such as: closest LMRs first, a predefined order of LMRs, "use what you have first" (first access LMRs to which you have keys from the previous partitions), etc. Second, there are numerous questions pertaining to the algorithms for acquiring quorum leases such as mentioned above. Which algorithms ensure that a quorum lease is finally achieved? How fast can this be done (in terms of messages/processing steps)? Can starvation be avoided when multiple partitions exist long enough? Is the current LMR model of operation sufficient enough for guaranteeing various properties of the negotiation algorithm (safety/liveness/fairness)? etc.

Other short-term tasks we intend to complete in the near future are benchmarking of the current implementation of FTS and porting FTS, as a proof-of-concept, to other platforms. FTS is currently implemented in Java over Linux using ORBacus [3] as an ORB and Ensemble [1, 12] as a GC toolkit. We would like to have it running on other operating systems such as Windows NT/2000 and Solaris, other ORBs, such as JacORB [2] and OpenORB [4], and other GC toolkits.
References


[17] OMG. Portable Interceptors. ptc/01-03-04.