Experiences, Strategies and Challenges in Building Fault-Tolerant CORBA Systems

Pascal Felber

Institut Eurecom
2229 Route des Cretes, BP 193
06904 Sophia Antipolis
France

pascal.felber@eurecom.fr

Priya Narasimhan

Electrical & Computer Engineering Department
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh
PA 15213, USA

priya@cs.cmu.edu

Abstract

After almost a decade since the introduction of the earliest reliable CORBA implementation, and despite the recent adoption of the Fault Tolerant CORBA (FT-CORBA) standard by the Object Management Group, CORBA is still not widely adopted as the preferred platform for building reliable distributed applications. Among the obstacles to FT-CORBA's widespread deployment are the complexity of the new standard, the lack of understanding in implementing and/or deploying reliable CORBA applications, and the fact that current FT-CORBA implementations are not readily applicable to real-world complex applications. In this paper, we candidly share our independent experiences as developers of two separate reliable CORBA infrastructures (OGS and Eternal), and as contributors to the FT-CORBA standardization process. Our intention is to reveal the intricacies, challenges and strategies in developing fault-tolerant CORBA systems, including our own. We provide an overview of the new FT-CORBA standard, and discuss its limitations and techniques for best exploiting it. We reflect on the difficulties that we encountered in building dependable CORBA systems, the solutions that we developed to address these challenges, and the lessons that we learned as a result. Finally, we highlight some of the open issues, such as non-determinism and partitioning, along with some solutions for resolving these issues.

Keywords: CORBA, FT-CORBA, fault tolerance, non-determinism, replication, recovery, OGS, Eternal

Pages: 34 (+ 1 in appendix)

Formatting: single column, double spaced, 12pt, Infocom style

Contact author: Pascal Felber, pascal.felber@eurecom.fr

I. INTRODUCTION

The integration of distributed computing with object-oriented programming leads to distributed object computing, where objects are distributed across machines, with client objects invoking operations on, and receiving responses from, remote server objects. Both the client's invocations, and the server's responses, are conveyed in the form of messages sent across the network. The Common Object Request Broker Architecture (CORBA) [31], was established by the Object Management Group, as a standard for distributed object computing.

CORBA uses a purely declarative language, the OMG Interface Definition Language (IDL), to define interfaces to objects. The IDL interfaces are subsequently mapped, through an IDL compiler provided by the CORBA implementor, onto specific programming languages. These IDL compilers conform to the OMG-standardized IDL-to-language mappings for C, C++, Java, Smalltalk, $etc.$ CORBA's language transparency implies that client objects need to be aware of only the IDL interface, and not the language-specific implementation, of a server object. CORBA's interoperability implies that a client object can interact with a server object, despite heterogeneity in their respective platforms and operating systems. CORBA's location transparency implies that client objects can invoke server objects, without worrying about the physical locations of the server objects. The key component of the CORBA model, the Object Request Broker (ORB), acts as an intermediary between the client and the server objects, and shields them from differences in platform, programming language and location.

Until recently, CORBA implementations had no standardized support for fault tolerance. Several research efforts were expended on remedying this deficiency. These early fault-tolerant implementations of CORBA adopted diverse approaches for reliability. Fundamentally, though, all of these approaches exploited some form of replication to protect the objects of the target CORBA application against faults.[1]

The various fault-tolerant CORBA implementations that resulted from either research or industrial efforts over the past decade can be broadly classified into:

- The *integration approach*, where the support for replication is integrated into the ORB (e.g., systems such as Electra [18], Orbix+Isis [17], and Maestro [37]),
- The *interception approach*, where the support for replication is provided transparently underneath the ORB (e.g., Eternal [24]), and
- The *service approach*, where the support for replication is provided primarily through a collection of

[1] Replication is a common strategy for roll-forward reliability; other strategies, such as transactions, can be used for roll-back reliability.

CORBA objects above the ORB (e.g., systems such as OGS [7], AQuA [5], DOORS [27], Newtop [20], FRIENDS [6], FTS [13] and IRL [19]).

Each of these systems has contributed significantly to our collective understanding of the problems of providing fault tolerance for CORBA applications. At the same time, each of these systems suffers from some inherent drawbacks (e.g., regular clients cannot interact with replicated servers in a fault-tolerant manner) and exposes different, non-standard APIs to the application, making development of portable fault-tolerant CORBA applications almost impossible.

More recently, the Object Management Group adopted a specification [29] for Fault-Tolerant CORBA (FT-CORBA)[2] that allows CORBA applications to be made more reliable through standardized APIs. While the specification is rather detailed, it does not fully address some of the complex issues faced by developers of real-world CORBA applications.

In this paper, we discuss the challenges that are commonly faced in developing fault-tolerant CORBA implementations, the strategies that can be used to address these challenges, and the insights that are gleaned as a result. Our wealth of experience – both as independent implementors of separate fault-tolerant CORBA implementations (Eternal and OGS, respectively), and as contributors to the FT-CORBA standard – allows us to reflect candidly and critically on the state-of-the-art and the state-of-practice in fault-tolerant CORBA, and on the significant challenges that remain to be resolved.

A. Replication

CORBA applications can be made fault-tolerant by replicating their constituent objects, and distributing these replicas across different processors in the network. The idea behind object replication is that the failure of a replica (or of a processor hosting a replica) of a CORBA object can be masked from a client because the other replicas can continue to provide the services that the client requires.

Replica consistency. Replication fails in its purpose unless the replicas are true copies of each other, both in state and in behavior. Strong replica consistency implies that the replicas of an object are consistent or identical in state, under fault-free, faulty and recovery conditions. Because middleware applications involve clients modifying a server's state through invocations, and servers modifying their clients' states through responses, the transmission of these invocations and responses becomes critical in maintaining consistent replication. One strategy for achieving this is to transmit all of the client's (server's) invocations (responses)

[2] In the remainder of the text, *fault-tolerant CORBA* refers to any system that provides reliability to CORBA applications, and *FT-CORBA* refers to the new OMG standard for reliable CORBA.

so that all of the server (client) replicas receive and, therefore, process the same set of messages in the same order. Another issue is that replication results in multiple, identical client (server) replicas issuing the same invocation (response), and these duplicate messages should not be delivered to the target server (client) as they might corrupt its state. Consistent replication requires mechanisms to detect, and to suppress, these duplicate invocations (responses) so that the target server (client) receives only one, non-duplicate, invocation (response).

All three approaches to fault-tolerant CORBA require the application to be deterministic, $i.e.$, any two replicas of an object, when starting from the same initial state and after processing the same set of messages in the same order, will reach the same final state. Mechanisms for strong replica consistency (ordered message delivery, duplicate suppression, $etc.$) along with the deterministic behavior of applications, enables effective fault tolerance so that a failed replica can be readily replaced by an operational one without losing any data or any computation.

Replication Styles. There are essentially two kinds of replication styles – active replication and passive replication [23]. With *active* replication, each server replica processes every client invocation, and returns the response to the client (of course, care must be taken to ensure that only one of these duplicate responses is actually delivered to the client). The failure of a single active replica is masked by the presence of the other active replicas that also perform the operation and generate the desired result. With *passive* replication, only one of the server replicas, designated the *primary*, processes the client's invocations, and returns responses to the client. With *warm passive* replication, the remaining passive replicas, known as *backups*, are preloaded into memory and are synchronized periodically with the primary replica, so that one of them can take over should the primary replica fail. With *cold passive* replication, however, the backup replicas are "cold", $i.e.$, not even running, as long as the primary replica is operational. To allow for recovery, the state of the primary replica is periodically checkpointed and stored in a log. If the existing primary replica fails, a backup replica is launched, with its state initialized from the log, to take over as the new primary. Both passive and active replication styles require mechanisms to support state transfer; for passive replication, the transfer of state occurs periodically from the primary to the backups, from the primary to a log, or from the log to a new primary; for active replication, the transfer of state occurs when a new active replica is launched and needs its state synchronized with the operational active replicas.

Object Groups. The integration, service and interception approaches are also alike in their use of the *object group* abstraction, where an object group represents a replicated CORBA object, and the group members

represent the individual replicas of the CORBA object. Object group communication is a powerful paradigm because it often simplifies the tasks of communicating with a replicated object, and of maintaining consistent replication. Groups were first introduced in the V-Kernel [3] as a convenient addressing mechanism, and were later extended to handle the replication of processes (rather than objects) in the Isis system [1]. The central idea behind group communication is to consider a set of processes or objects as a logical group, and to provide primitives for sending messages simultaneously to the group as a whole, usually with various ordering guarantees on the delivered messages. A group constitutes a logical addressing facility because messages can be issued to groups without requiring any knowledge of the number, the identities, or even the locations of the individual members of the group.

With an object group being equivalent to a replicated CORBA object, group communication can be used to maintain the consistency of the states of the object's replicas. Reliable ordered multicast protocols often serve as concrete implementations of, and are therefore synonymous with, group communication systems. For this reason, one or more of the various fault-tolerant CORBA systems described in this section employ totally-ordered reliable multicast group communication toolkits to facilitate consistent replication.

Relevant Non-CORBA Systems. This discussion of replication would not be complete without a brief overview of reliable systems that preceded, and paved the way for, fault-tolerant CORBA. The Delta-4 system [33] provided fault tolerance in a distributed Unix environment through the use of an atomic multicast protocol to tolerate crash faults at the process level. Delta-4 supported active replication and passive replication, as well as hybrid semi-active replication. The Arjuna system [32] used object replication together with an atomic transaction strategy to provide fault tolerance. Arjuna supported active replication, coordinator-cohort passive replication and single-copy passive replication. These systems, and the insights that they provided, have contributed greatly to our understanding of fault tolerance. It is no surprise that almost every fault-tolerant CORBA system embodies principles that are derived from one or the other of these systems.

II. Existing Fault-Tolerant CORBA Systems

Initial efforts to enhance CORBA with fault tolerance leaned towards the integration approach, with the fault tolerance mechanisms embedded *within* the ORB itself. With the advent of Common Object Services in the CORBA standard, other research efforts adopted the service approach, with the fault tolerance support provided by service objects *above* the ORB. Yet another strategy, the interception approach, allowed the transparent insertion of fault tolerance mechanisms *underneath* the ORB. The three approaches are similar in

their use of object replication to provide fault tolerance.

The underlying system model for all three approaches is an asynchronous distributed system, in which processors communicate via messages over a local area network that is completely connected. Communication channels are not assumed to be FIFO or authenticated, but the network is assumed not to partition. A processor receives all of its own messages. The system is asynchronous in that no bound can be placed on the time required for a computation or for the communication of a message. Processors have access to local clocks, but these clocks are not necessarily synchronized.

As concerns the fault model, the system is subject to communication, processor, and object faults. Communication between processors is unreliable and, thus, messages may need to be retransmitted. Processors, processes and objects are subject to crash faults, and thus, might require recovery and re-instatement to correct operation.

A. The Integration Approach

The integration approach to providing new functionality to CORBA applications involves modifying the ORB to provide the necessary fault tolerance support. The CORBA standard requires every compliant ORB to support the TCP/IP-based Internet Inter-ORB Protocol (IIOP), but the addition of group communication support directly into the ORB is likely to involve replacing IIOP by a proprietary group communication protocol and thus violate this requirement. The resulting modified, but fault-tolerant, ORB may therefore be non-compliant with the CORBA standard.

However, because the fault tolerance mechanisms form an intrinsic part of the ORB, they can be implemented so that the application's interface to the ORB (and the behavior that the application expects of the ORB) remains unchanged. Thus, an integration approach to providing fault tolerance for CORBA implies that the replication of server objects can be made transparent to the client objects because the fault tolerance mechanisms are part of the ORB. Furthermore, the details of the replica consistency mechanisms are buried within the ORB, and can be hidden from the application programmer.

Electra. Developed at the University of Zurich, Electra [18] is the earliest implementation of a fault-tolerant CORBA system, and consists of a modified ORB that exploits the reliable totally ordered group communication mechanisms of the Horus toolkit [36] to maintain replica consistency. As shown in Figure 1(a), adaptor objects that are linked into the ORB (and, therefore, implicitly into the CORBA application) convert the application's/ORB's messages into multicast messages of the underlying Horus toolkit. In Electra, the Basic

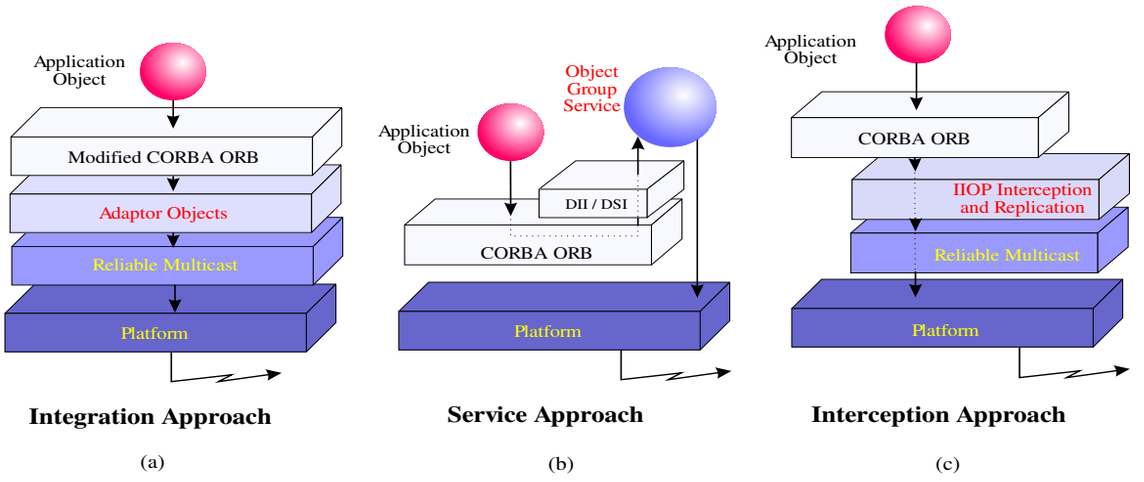

Fig. 1. Different approaches to fault-tolerant CORBA.

Object Adapter (an ORB component that has been rendered obsolete by the Portable Object Adapter of the CORBA 2.x standard) of the CORBA 1.x standard is enhanced with mechanisms for creating and removing replicas of a server object, and for transferring the state to a new server replica.

With Electra's use of the integration approach, a CORBA client hosted by Electra can invoke a replicated server object just as it would invoke a single server object, without having to worry about the location, the number, or even the existence, of the server replicas.

Orbix+Isis. Developed by Iona Technologies, Orbix+Isis [17] was the first commercial offering in the way of fault tolerance support for CORBA applications. Like Electra, Orbix+Isis involves significant modification to the internals of the ORB to accommodate the use of the Isis toolkit [1] from Isis Distributed Systems for the reliable ordered multicast of messages.

With Orbix+Isis, the implementation of a CORBA server object must explicitly inherit from a base class. Two types of base classes are provided – an *Active Replica* base class that provides support for active replication and hot passive replication, and an *Event Stream* base class that provides support for publish-subscribe applications.

The replication of server objects can be made transparent to the client objects. Orbix-specific smart proxies can be used on the client side to collect the responses from the replicated server object, and to use some policy (delivering the first received response, voting on all received responses, $etc.$) in order to deliver a single response to the client object.

Maestro Replicated Updates ORB. Developed at Cornell University, Maestro [37] is a CORBA-like implementation of a distributed object layer that supports IIOP communication and that exploits the Ensemble group communication system [35]. The ORB is replaced by an IIOP Dispatcher and multiple request managers that are configured with different message dispatching policies. One of these request managers, called the Replicated Updates request manager, supports the active replication of server objects. "Smart" clients have access to compound Interoperable Object References[3] (IORs) consisting of the enumeration of the IIOP profiles ($i.e.$, the addresses) of all of the replicas of a server object. A "smart" client connects to a single server replica and, in the event that this replica fails, can re-connect to one of the other server replicas using the addressing information in the compound IOR.

In the typical operation of Maestro, a client object running over a commercial ORB uses IIOP to access a single Maestro-hosted server replica, which then propagates the client's request to the other server replicas through the messages of the underlying Ensemble system. However, the server code must be modified to use the facilities that the request managers of Maestro provide. Maestro's emphasis is on the use of IIOP and on providing support for interworking with non-CORBA legacy applications, rather than on strict adherence to the CORBA standard. Thus, Maestro's replicated updates execution style can be used to add reliability and high availability to client/server CORBA applications in settings where it is not feasible to make modifications at the client side.

B. The Service Approach

The service approach to extending CORBA with new functionality involves providing the enhancements through a new service, along the lines of the existing Common Object Services [28] that form a part of the CORBA standard. Because the new functionality is provided through a collection of CORBA objects entirely above the ORB, the ORB does not need to be modified and the approach is CORBA-compliant. However, to take advantage of the new service, the CORBA application objects need to be explicitly aware of the service objects. Thus, it is likely that application code requires modification to exploit the functionality of the new CORBA service.

Using this approach, fault tolerance can be provided as a part of the suite of CORBA Services. Of course, because the objects that provide reliability reside above the ORB, every interaction with these objects must

[3] An Interoperable Object Referencey (IOR) is a stringified form of a reference to a CORBA object, and can contains one or more profiles. Each profile contains sufficient information to contact the object using some protocol, usually TCP/IP; this information often includes the host name, port number, and object key associated with the CORBA object.

necessarily pass through the ORB, and will thus incur the associated performance overheads.

Distributed Object-Oriented Reliable Service (DOORS). The Distributed Object-Oriented Reliable Service (DOORS) [27] developed at Lucent Technologies adds support for fault tolerance to CORBA by providing replica management, fault detection, and fault recovery as service objects above the ORB. DOORS focuses on passive replication and is not based on group communication and virtual synchrony. It also allows the application designer to select the replication style (cold passive and warm passive replication), degree of reliability, detection mechanisms and recovery strategy.

DOORS consists of a WatchDog, a SuperWatchDog and a ReplicaManager. The WatchDog runs on every host in the system and detects crashed and hung objects on that host, and also performs local recovery actions. The centralized SuperWatchDog detects crashed and hung hosts by receiving heartbeats from the WatchDogs. The centralized ReplicaManager manages the initial placement and activation of the replicas and controls the migration of replicas during object failures. The ReplicaManager maintains a repository that contains, for each object in the system, the number of replicas, the hosts on which they are running, the status of each replica and the number of faults seen by the replica on a given host. This repository, which forms part of the state of the ReplicaManager, is periodically checkpointed. DOORS employs libraries for the transparent checkpointing [39] of applications.

Object Group Service (OGS). Developed at the Swiss Federal Institute of Technology, Lausanne, the Object Group Service (OGS) [10], [7] consists of service objects implemented above the ORB that interact with the objects of a CORBA application to provide fault tolerance to the application. OGS is comprised of a number of sub-services, with interfaces specified using OMG IDL, implemented on top of off-the-shelf CORBA ORBs. The multicast sub-service provides for the reliable unordered multicast of messages destined for the replicas of a target server object; the messaging sub-service provides the low-level mechanisms for mapping these messages onto the transport layer; the consensus sub-service imposes a total order on the multicast messages; the membership sub-service keeps track of the composition of obect groups; finally, the monitoring sub-service detects crashed objects. Each of these sub-services is independent and is itself implemented as a collection of CORBA objects.

To exploit the facilities of the OGS objects, the replicas of a server object must inherit from a common IDL interface that permits them to join or leave the group of server replicas. Thus, in order to be replicated, the server objects must be modified. This interface also provides methods that allow the OGS objects to transfer the state of the replicated server objects, as needed, to ensure replica consistency.

OGS provides a client object with a local proxy for each replicated server with which the client communicates. The server's proxy on the client side and the OGS objects on the server side are together responsible for the mapping of client requests and server responses onto multicast messages that convey the client's request to the server replicas. The client establishes communication with the replicas of a server object by binding to an identifier that designates the object group representing all of the server replicas. The client can then direct its requests to the replicated server object using this object group identifier. Once a client is bound to a server's object group, it can invoke the replicated server object as if it were invoking a single unreplicated server object. However, because the client is aware of the existence of the server replicas, and can even obtain information about the server object group, the replication of the server is not necessarily transparent to the client. Also, with this approach, a CORBA client needs to be modified to bind, and to dispatch its requests, to a replicated CORBA server.

Newtop Object Group Service. Developed at the University of Newcastle, the Newtop [20] service provides fault tolerance to CORBA using the service approach. While the fundamental ideas are similar to OGS (described in Section II-B), Newtop has some key differences.

Newtop allows objects to belong to multiple object groups. Of particular interest is the way the Newtop service handles failures due to partitioning – support is provided for a group of replicas to be partitioned into multiple sub-groups, with each sub-group being connected within itself. Total ordering continues to be preserved within each sub-group. No mechanisms are provided, however, to ensure consistent remerging of the sub-groups once communication is reestablished between them.

IRL and FTS. The Interoperable Replication Logic (IRL) [19] also provides fault tolerance for CORBA applications through a service approach. One of the aims of IRL is to uphold CORBA's interoperability by supporting a fault-tolerant CORBA application that is composed of objects running over implementations of ORBs from different vendors. IRL aims to provide the client-side replication support required by the new FT-CORBA standard.

Like IRL, FTS [13] aims to provide the client-side replication support required by the new FT-CORBA standard. In addition, FTS aims to provide some support for network partitioning by imposing a primary partition model on the application in the event that the system is partitioned into disconnected components. Both IRL and FTS were developed after the adoption of the FT-CORBA standard.

The AQuA Framework. Developed jointly by the University of Illinois at Urbana-Champaign and BBN Technologies, AQuA [5] is a framework for building fault-tolerant CORBA applications. AQuA employs the Ensemble/Maestro [35], [37] toolkits, and comprises the Quality Objects (QuO) runtime, and the Proteus dependability property manager [34]. Based on the user's QoS requirements communicated by the QuO runtime, Proteus determines the type of faults to tolerate, the replication policy, the degree of replication, the type of voting to use and the location of the replicas, and dynamically modifies the configuration to meet those requirements. The AQuA gateway translates a client's (server's) invocations (responses) into messages that are transmitted via Ensemble; the gateway also detects and filters duplicate invocations (responses). The gateway handlers contain monitors, which detect timing faults, and voters, which either accept the first invocation/response or perform majority voting on the invocations/responses from the object replicas.

AQuA provides mechanisms for majority voting at the application object level, to detect an incorrect value of an invocation (response) from a replicated client (server). However, in order for majority voting to be effective for applications that must tolerate arbitrary faults, more stringent guarantees are required from the underlying multicast protocols than are provided by the underlying group communication system, which tolerates only crash faults.

FRIENDS. The FRIENDS [6] system aims to provide mechanisms for building fault-tolerant applications in a flexible way through the use of libraries of meta-objects. Separate meta-objects can be provided for fault tolerance, security and group communication. FRIENDS is composed of a number of subsystems, including a fault tolerance subsystem that provides support for object replication and detection of faults. A number of interfaces are provided for capturing the state of an object to stable storage, and for transmitting the state of the primary replica to the backup replicas in the case of passive replication.

C. The Interception Approach

The interception approach to extending CORBA with new functionality involves providing fault tolerance transparently through the use of an interceptor, which is a software component that can attach itself to existing pre-compiled and pre-linked software. The interceptor can then contain additional code to modify the behavior of the application, without the application or the ORB being ever aware of the interceptor's existence or operation. The transparency implies that fault tolerance can be provided for binary or executable code because neither the ORB nor the application needs to be modified, re-compiled or re-linked.

The disadvantage is that if the interception mechanisms are specific to the operating system, as is often the

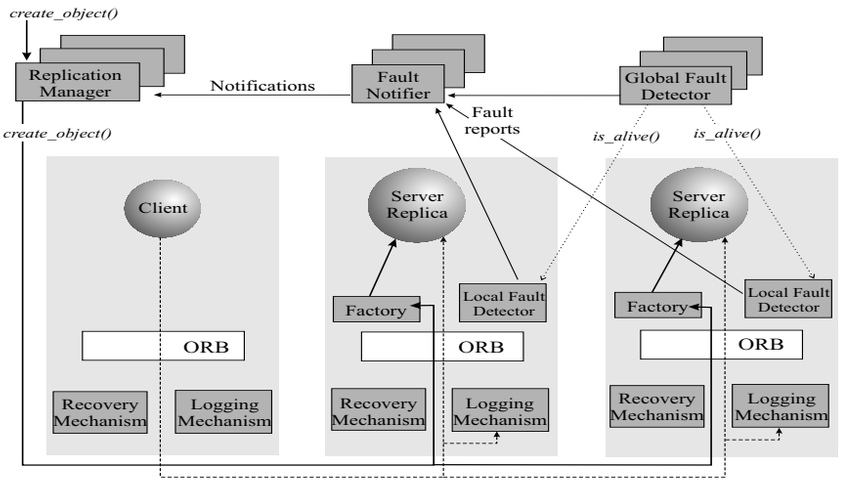

Fig. 2. Architectural overview of the Fault-Tolerant CORBA (FT-CORBA) standard.

case, then, the interceptor needs to be ported to every operating system that is intended to run the CORBA application. The Portable Interceptors [30] that CORBA currently offers are the result of standardizing such useful interception "hooks" in order to avoid the effort of porting them to various operating systems or ORBs.

Eternal. Developed at the University of California, Santa Barbara, the Eternal system [24], [21] exploits the interception approach to provide fault tolerance for applications running over commercial off-the-shelf implementations of CORBA. The mechanisms implemented in different parts of the Eternal system work together efficiently to provide strong replica consistency without requiring the modification of either the application or the ORB. The current implementation of Eternal works with both C++ and Java ORBs, including VisiBroker, Orbix, Orbacus, ILU, TAO, e*ORB, omniORB2 and CORBAplus.

Different replication styles – active, cold passive, warm passive and hot passive replication – of both client and server objects are supported. To facilitate replica consistency, the Eternal system conveys the IIOP messages of the CORBA application using the reliable totally ordered multicast messages of the underlying Totem system [22]. The Eternal Replication Manager replicates each application object, according to user-specified fault tolerance properties (such as the replication style, the checkpointing interval, the fault monitoring interval, the initial number of replicas, the minimum number of replicas, $etc.$), and distributes the replicas across the system. The Eternal Interceptor captures the IIOP messages (containing the client's requests and the server's replies), which are intended for TCP/IP, and diverts them instead to the Eternal Replication Mechanisms for multicasting via Totem. The Eternal Replication Mechanisms, together with the Eternal Logging-Recovery Mechanisms, maintain strong consistency of the replicas, detect and recover from faults, and sustain operation in all components of a partitioned system, should a partition occur. Gateways [26]

allow unreplicated clients that are outside the system to connect to, and exploit the services of, replicated server objects. Eternal also provides for controlled thread scheduling to eliminate the non-determinism that multithreaded CORBA applications exhibit.

Eternal tolerates communication faults, including message loss and network partitioning, and processor, process, and object faults. Eternal can also tolerate arbitrary faults by exploiting protocols such as secure reliable multicast protocols, with more stringent guarantees than are provided by Totem. To tolerate value faults in the application, Eternal uses active replication with majority voting [25] applied on both invocations and responses for every application object.

D. The FT-CORBA Standard

A standard specification of Fault-Tolerant CORBA (FT-CORBA) [29] has been recently formally adopted by the Object Management Group (OMG). This specification describes minimal fault-tolerant mechanisms to be included in any CORBA implementation, as well as interfaces for the more advanced facilities provided by a fault-tolerant CORBA implementation. The specification also includes support for configuring fault tolerance (through several fault tolerance properties). FT-CORBA implementors are free to use proprietary mechanisms (such as reliable multicast protocols) for their actual implementation, as long as the resulting system complies with the interfaces defined in the specification, and the behavior expected of those interfaces.

Several groups involved in developing previous fault-tolerant CORBA implementations (namely [27], [21], [10]) have contributed to, and heavily influenced, the FT-CORBA specification, and therefore this specification builds on experiences from some of the fault-tolerant CORBA systems described earlier in Section II. Due to the fundamental differences between these systems and diverging goals of the industrial and academic participants involved in the standardization process, the resulting specification evolved into a very general, but also complex, replication framework. As an example, the service specification makes it possible to implement fault tolerance through such different mechanisms as group communication and replicated databases. The client-side mechanisms to be included in all CORBA implementations — regardless of whether they implement FT-CORBA or not — have however been kept minimal. They basically specify object references that can contain multiple profiles, each of which designates a replica (multi-profile IORs), and simple rules for iterating through the profiles in case of failure. These mechanisms ensure that unreplicated clients can interact with replicated FT-CORBA-supported servers in a fault-tolerant manner. We know of two implementations of FT-CORBA, the latest versions of Eternal [21] and DOORS [27], as well as a few CORBA implementations,

such as IRL [19] and FTS [13], that already include support for FT-CORBA's basic client-side mechanisms.

Figure 2 shows the architecture of the FT-CORBA specification. The Replication Manager handles the creation, deletion and replication of both the application objects and the infrastructure objects. The Replication Manager replicates objects, and distributes the replicas across the system. Although each replica of an object has an individual object reference, the Replication Manager fabricates an object group reference for the replicated object that clients use to contact the replicated object. Note that all the replicas of a given object are expected to be deployed on the same FT-CORBA infrastructure, i.e., heterogeneity is not supported within groups. The Replication Manager's functionality is achieved through the Property Manager, the Generic Factory and the Object Group Manager.

The Property Manager allows the user to assign values to an object's fault tolerance properties, including *Replication Style* (stateless, actively replicated, cold passively replicated or warm passively replicated),*Membership Style* (addition, or removal, of an object's replicas is application-controlled or infrastructure-controlled), *Consistency Style* (replica consistency, including recovery, checkpointing, logging, $etc.$, is application-controlled or infrastructure-controlled), *Factories* (objects that create and delete the replicas of the object), *Initial Number of Replicas* (the number of replicas of an object to be created initially), *Minimum Number of Replicas* (the number of replicas of the object that must exist for the object to be sufficiently protected against faults), *Checkpoint Interval* (the frequency at which the state of an object is to be retrieved and logged for the purposes of recovery), *Fault Monitoring Style* (the object is monitored by periodic "pinging" of the object, or, alternatively, by periodic "i-am-alive" messages sent by the object, i.e., push or push monitoring), *Fault Monitoring Granularity* (the replicated object is monitored on the basis of an individual replica, a location or a location-and-type), and *Fault Monitoring Interval* (the frequency at which an object is to be "pinged" to detect if it is alive or has failed).

The Generic Factory allows users to create replicated objects in the same way that they would create unreplicated objects. The Object Group Manager allows users to control directly the creation, deletion and location of individual replicas of an application object, and is useful for expert users who wish to exercise direct control over the replication of application objects.

The Fault Detector is capable of detecting host, process and object faults. Each application object inherits a `Monitorable` interface to allow the Fault Detector to determine the object's status. The Fault Detector communicates the occurrence of faults to the Fault Notifier. The Fault Detectors can be structured hierarchically, with the global replicated Fault Detector triggering the operation of local fault detectors on each

Approach	Advantages	Disadvantages	Systems
Integration	– Transparent to the application	– Proprietary, modified ORB – Porting required for every new ORB	Electra, Orbix+Isis
Service	– CORBA-compliant – Can exploit CORBA's interoperability to work with any ORB	– Not always transparent to the application	OGS, DOORS, FRIENDS, Newtop, AQuA, FTS, IRL
Interception	– Transparent to the application – Can exploit CORBA's interoperability to work with any ORB	– Interceptor needs to be ported to every new operating system	Eternal
FT-CORBA Standard	– Standardized, configurable support	– Describes interfaces and properties, leaving implementation details open – Requires extensions to the standard ORB core	Eternal, DOORS

TABLE I
COMPARISON OF DIFFERENT APPROACHES TO FAULT-TOLERANT CORBA.

processor. Any faults detected by the local fault detectors are reported to the global replicated Fault Notifier.

On receiving reports of faults from the Fault Detector, the Fault Notifier filters them to eliminate any duplicate reports. The Fault Notifier then distributes fault event notifications to all of the objects that have subscribed to receive such notifications. The Replication Manager, being a subscriber of the Fault Notifier, receives reports of faults that occur in the system, and can initiate appropriate actions to enable the system to recover from faults.

III. CRITICAL LOOK AT FAULT-TOLERANT CORBA SYSTEMS

There are specific issues which bring out the differences amongst the three approaches and the FT-CORBA standard. In this section, we take a critical look at the integration, service and interception approaches, side by side with the FT-CORBA standard, in order to identify their respective challenges and limitations.

A. Server-Side and Client-Side Transparency

We illustrate the issues in server-side and client-side transparency when implementing fault-tolerant CORBA applications using the following example of a simplified banking application. A bank is a CORBA object that is used to maintain a large number of accounts. Each account is identified by a unique account number, and contains the name of the account holder as well as the current balance. An account offers two operations for deposits and withdrawals. For simplicity, we do not handle exceptions ($e.g.$, overdraft). The code of this application is found in appendix.

CORBA promotes transparency by allowing the CORBA programmer to write the bank application without worrying about issues such as differences in the client's and server's location in the system (location transparency), differences in the client's and server's operating systems (interoperability), and differences in the client's and server's programming languages (language transparency). A client first obtains a reference to

an account from a naming service or by any other mean; then it can perform deposits and withdrawals on the associated account, as if the account were a local object.

Because the bank object is a critical resource, it should be replicated and distributed across different processors in the distributed system. Replication transparency hides the use of replication and of fault tolerance from the application programmer, by providing him/her the illusion that the invocations (responses) are issued to, and originate from, single objects. Transparency is clearly desirable because it relieves the application programmer of the burden of dealing explicitly with difficult issues such as fault tolerance, and also allows him/her to continue programming applications as before. Informally, we can distinguish between transparency from two different perspectives:

- **Client-side transparency:** Here, the client is unaware of the server's replication. Thus, the client code does not need to be modified to communicate with the now-replicated server. The main difficulty in achieving client-side transparency is to make the replicated server's group reference ($i.e.$, references containing addresses of all of the server replicas) appear the same as the usual unreplicated server's reference to the client application. Fault-tolerant CORBA systems deal with this in different ways. The integration approach (Electra and Orbix+Isis) uses custom object reference types, and requires clients to execute on top of the proprietary reliable ORBs. The interception approach (Eternal) transparently maps CORBA's normal interoperable object references (IORs) to implementation-specific group references that are hidden from the client application. The service approach (OGS and IRL) let the application code explicitly deal with group references, or transparently invoke replicated servers through a generic gateway that maps normal object references to group references. With the FT-CORBA standard, group-specific information can be embedded into object references using the IIOP profile mechanism; however, this information is not intended to be exposed to the client application. The FT-CORBA group references are intended to be consumed solely by client-side ORBs that have been enhanced with fault tolerance mechanisms. Although the original client code (which is not replication-aware) can generally be used with no modification to invoke a replicated server, it might need to be compiled/linked with different libraries in order to obtain the respective support provided by the interception approach, the service approach, or the FT-CORBA standard.
- **Server-side transparency:** Here, the server objects are unaware of their own, and of each other's, replication. Thus, the server code does not need to be modified to support its own replication, or the replication of other servers. Server-side transparency is far more difficult to achieve than client-side replication. If the server is replicated (using either active or passive replication), then, there needs to exist some way of retriev-

ing and transferring the state of a replica, for recovery and for consistent replication. Because an object's state consists of the values of its application-level data structures, there must be some provision for extracting these values from the application. FT-CORBA provides for this through the additional standardized interfaces that every CORBA object must inherit if the object is to be replicated. The code of the bank account server must thus be modified in order to be replicated: the `Account` object class now inherits from the `Updateable` and (indirectly) the `Checkpointable` interfaces that define operations for the retrieval and the assignment of the partial or the complete state, respectively, of the object. The complete state refers to the entire state of an account object (account number, account holder, current balance) while the partial state (also known as an update or a state increment) refers to only that part (most likely, the balance) of the object's state that has been modified since the last state snapshot was taken. Although FT-CORBA specifies the names and method signatures of these interfaces, the integration, service and interception approaches use some variant of these interfaces because $every$ fault-tolerant CORBA system that supports stateful servers with consistent replication requires each server to support some kind of interface for state retrieval and assignment.

Thus, true server-side transparency ($i.e.$, absolutely no modifications to the server code) is impossible to achieve as long as the server interface must support these additional interfaces for state retrieval and assignment. However, server-side transparency can generally be achieved when CORBA objects are stateless, or if there exist other ways of retrieving and assigning an object's state without using an IDL interface. For instance, it is possible to take snapshots of the operating systems' layout of an entire process' state, and to transfer and to assign these snapshots to replicas of the process; in such cases, there is no need for the retrieval, assignment or transfer of application-level state. However, this kind of pickling mechanism [4] poses problems in terms of transferring local pointers, handling the heterogeneity of operating systems, $etc.$ Thus, for all practical purposes, no fault-tolerant CORBA infrastructure (regardless of whether it uses the interception, integration or service approaches, or the FT-CORBA standard) ever fully achieves server-side transparency. The problem discussed in Section III-G is a further impediment to server transparency.

Note that objects that play the role of both client and server ($i.e.$, they act as a server to one set of objects, and as a client to possibly a different set of objects) might require support for both client-side and server-side transparency. Such dual-role objects need support for client replication, in addition to the server replication that most fault-tolerant systems traditionally consider. This means that when such an object is recovered, both its client-side and its server-side state needs to be re-instated consistently. Unfortunately, both CORBA and FT-CORBA are server-centric, and support only the notions of handling server state through server-side

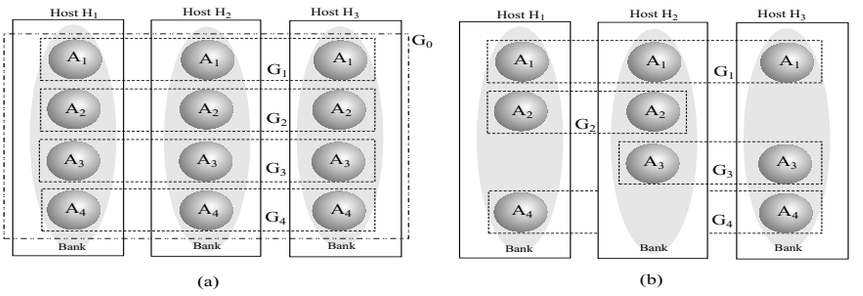

Fig. 3. Processors hosting replicas of `Account` objects (A_i is the *i*th distinct replicated `Account` object) in different configurations: (a) fully overlapping groups and (b) partially overlapping groups.

interfaces. In order to replicate dual-role objects, the `Checkpointable` or the `Updateable` interfaces need to be extended, to extract, and to assign, the client-side state. Other issues with dual-role objects are discussed further in Section III-D.

Finally, it has been argued that even when transparency is technically achievable, it requires generic protocols that behave in a conservative way and that cannot perform optimizations, thereby resulting in poor performance [38]. A promising approach to cope with this problem, discussed in [11], consists in using semantic knowledge of the application to determine the optimal protocols necessary to guarantee replica consistency. Semantic information (e.g., knowledge that operations are read-only, deterministic, or commutative) is not an intrinsic part of the application code, but can be specified when the application is deployed. This additional information can then be conveyed to the underlying fault-tolerant CORBA infrastructure in order to facilitate various tasks like transparent load balancing, caching, or low-level protocol optimizations.

B. Object vs. Processes

Group communication toolkits have traditionally dealt with process groups; if a process fails, then it is removed from all the groups to which it belongs. With FT-CORBA, the fundamental unit of replication is the object, and as such, groups deal with objects rather than processes. Since a CORBA application typically manages a large number of objects, this may lead to the creation and management of a large number of object groups. This is known as the group proliferation problem [15].

For instance, in our CORBA bank application example (see Section III-A), there can exist multiple, different `Account` objects in the system. When replicated, each distinct `Account` object becomes associated with a unique object group, with all of the resulting object groups distributed across the same set of processors (see Figure 3(a)). In particular, if any one of the processors fails, all of the `Account` replicas hosted by

that processor can be considered to have failed. The fault-tolerant CORBA infrastructure must then update all of the associated object groups as part of a membership-change protocol; this might require notifying each individual `Account` replica of the new membership of its associated object group. Thus, the failure of a single processor can lead to multiple object group membership-change messages propagating through the system.

This problem can sometimes be avoided by having the underlying fault-tolerant infrastructure detect object groups that span the same set of processors, and treat them as logical groups that all map onto a single physical group. This approach is, however, not readily applicable when groups partially overlap, as shown in Figure 3(b). This is often the case when replicas are created and hosted across a subset of a pool of processors. This problem can be addressed by using a group communication protocol that supports sub-group membership.

The conflict between objects and processes results from the mismatch between object-oriented computing, which promotes a small granularity for application components, and fault-tolerant distributed computation, which benefits from coarse components. The developer can generally architect his/her application to avoid this problem. For instance, the bank application could be modified so that accounts are not represented by CORBA objects, and so that the bank is the only entity that is represented by an object group.

Another aspect of the mismatch between objects and processes is the fact that CORBA applications are often not pure object-oriented programs, $i.e.$, the state of an object might depend on some other entity ($e.g.$, global variables) that is outside the object, but nevertheless within the process containing the object. In fact, a CORBA object's "real" state can be considered to be distributed in three different places: (i) application-level state, consisting of the values of data structures within the CORBA object, (ii) ORB/POA-level state, consisting of "pieces" of state within the ORB and the Portable Object Adapter (POA) that affect, and are affected by, the CORBA object's behavior, $e.g.$, the last-seen outgoing IIOP request identifier, and (iii) infrastructure-level state, consisting of "pieces" of state within the fault-tolerant CORBA infrastructure that affect, and are affected by, the CORBA object's behavior, $e.g.$, the list of connections/clients for the object, the last-seen incoming/outgoing message identifier for duplicate detection.

Because a CORBA object's state is not contained entirely within the object, other parts of the object's process might need to be considered during the retrieval, assignment and transfer of the object's state. When a new replica of the CORBA object is launched, all three pieces of state – the application-level, the ORB/POA-level and the infrastructure-level state – needs to be extracted from an operational replica and transferred to

the recovering/new replica.

Application-level state is possibly the easiest to obtain because it can be extracted through the `Checkpointable` or `Updateable` interfaces. ORB/POA-level state is far more difficult to obtain because CORBA standardizes on interfaces and not on ORB implementations, which means that ORBs can differ widely in their internals. Furthermore, ORB vendors tend to regard (and also want their users to regard) their ORBs as stateless black-boxes and are reluctant to reveal the details of their proprietary mechanisms. Some of our biggest challenges in building strongly consistent fault-tolerant CORBA systems lie in deducing the ORB/POA-level state (with or without the assistance of the ORB vendor), and in retrieving, transferring and assigning this state correctly. Infrastructure-level state, although entailing additional mechanisms within the fault-tolerant infrastructure, is relatively easy for the fault-tolerant CORBA developer to deduce and to maintain.

Unfortunately, the "leakage" of the object's state into its containing process, through the ORB/POA-level and the infrastructure-level state, cannot be fully avoided, given the current state-of-the-art in ORB implementations. Because the ORB and the POA handle all connection and transport information on behalf of a CORBA object that they support, the ORB and the POA necessarily maintain some information for the object. This implies that there really are no stateless objects – a CORBA object with no application-level state will nevertheless have associated ORB/POA-level state. This implementation-dependent nature of the ORB/POA-level state means that different replicas of the same object cannot be hosted on ORBs from different vendors ($i.e.$, it is not possible to have a two-way replicated object with one replica hosted on ORB X, and the other replica hosted over ORB Y from a different vendor) because no assurances can be provided on the equivalence of the ORB/POA-level states of the respective ORBs. For all practical purposes, a strongly consistent replicated object must have all of its replicas running on an ORB from the same ORB vendor. Another consequence of the vendor-dependence of ORB/POA-level state is that a fault-tolerant CORBA developer must be fully aware of the internal, hidden differences across diverse ORBs, and must be able to deduce and handle this state for each new target ORB.

C. Interaction with Unreplicated Objects

Traditional group communication systems often assume a homogeneous environment, where all of the application's components are executing on top of the same fault-tolerant infrastructure. Ideally, a CORBA application that needs fault tolerance should have all of its components made fault-tolerant over the same

reliable CORBA infrastructure. In practice, however, distributed systems often need to obtain services from external or legacy components that do not necessarily have any support for replication or fault tolerance. In some cases, the fault-tolerant CORBA developer might not even have access to these external entities, $e.g.$, databases, web-based clients running behind firewalls. Thus, real-world fault-tolerant (replicated) applications will need to interoperate seamlessly with non-fault-tolerant (unreplicated) entities. These unreplicated objects must be able to invoke, as well as respond to, replicated objects.

Because of this interoperability requirement, the complexity of the fault-tolerant infrastructure increases. For instance, when receiving an IIOP request from an unreplicated client, the fault-tolerant CORBA infrastructure for an actively replicated server now needs to relay the request to all of the server replicas, and to ensure that only one reply is returned to the client. Similarly, when an actively replicated client issues a request to an unreplicated server, only one invocation must be seen by the unreplicated server, while the response must be received by all of the active client replicas.

Fault-tolerant CORBA infrastructures typically use gateways to alleviate the complexity of interfacing replicated CORBA objects with unreplicated objects or with ORBs from other vendors, as described in [26]. The part of the application or the system that the fault-tolerant CORBA infrastructure is responsible for supporting and rendering fault-tolerant, is referred to as a *fault-tolerant domain*. The role of a gateway is to bridge non-fault-tolerant clients/servers into a fault-tolerant domain. The gateway translates an unreplicated client's regular IIOP requests into the reliable multicast requests expected within the fault-tolerant domain, and vice-versa. Gateways can also be used to manage loosely-replicated, or weakly consistent, CORBA servers without the need for reliable protocols [8]. Another use for gateways is to bridge two fault-tolerant CORBA infrastructures, where each uses different mechanisms or different underlying reliable multicast protocols

Support for heterogeneity in FT-CORBA has a price, as end-to-end reliability cannot be guaranteed when unreplicated or non-fault-tolerant entities are involved. Consider, for instance, the case of a replicated `Payroll` object invoking the unreplicated version of the CORBA `Account` object in our bank example (see Section III-A). Note that the unreplicated `Account` object is outside the fault-tolerant domain, which means that we have no way of keeping track of which invocations/responses the `Account` object has received/processed, or whether a message is a duplicate of a previously received one. Suppose that the primary `Payroll` replica issues an IIOP request to the `Account` object (say, to deposit a pay-check) and, then, the primary replica fails. The new primary replica that takes over has no way of knowing (without manual

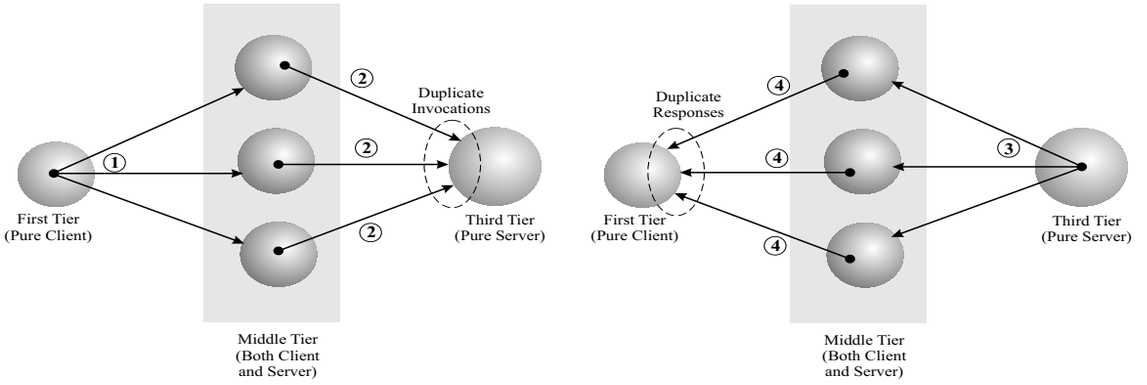

Fig. 4. Duplicate invocations and duplicate responses with an actively replicated middle tier.

intervention, through the payroll department calling up a bank officer) whether the unreplicated `Account` object ever received the invocation and whether it ever returned a response; undecided, the `Payroll` object might re-issue the request, resulting in possibly two pay-check deposits! Because the unreplicated `Account` object lacks the fault-tolerant infrastructural support for duplicate detection and suppression, there is no way for the `Account` object to report the erroneous duplicate deposits (it is doubtful whether the account holder would report this, either). A similar problem arises if an unreplicated `Payroll` object receives a reply from a replicated `Account` object.

Despite these issues, gateways are useful in many scenarios where fault-tolerant CORBA applications must necessarily deal with clients outside of their reach. However, in the interests of strong replica consistency, it is recommended to deploy all of the clients and servers of a CORBA application over the same fault-tolerant infrastructure, for reasons of both reliability and efficiency.

D. Multi-Tiered Applications

In a distributed object system, a component is often not restricted to either a pure client or a pure server role, but could act as a client for one operation and a server for another (in this sense, the terms “client” and “server” do not refer to the entities themselves, but rather to the roles that these entities play in an operation). This typically happens in a multi-tiered application, where middle-tier objects (also known as application servers or business logic) mediate interactions between front-end pure clients and back-end pure server systems. When, as a part of processing an incoming invocation, a server object acts as a client and, in turn, invokes another server object, the second invocation is referred to as a *nested/chained operation*. In the special case where the second server object is identical to the first server object, $i.e.$, the server object invokes itself, the nested invocation is also called a *callback operation*.

From the viewpoint of fault tolerance, providing support for nested invocations between distributed objects poses additional complications. Informally, the reason is that an invocation issued by a replicated client object to a remote server must be processed exactly once despite the failure of a client replica. Consider a three-tier application where, for the sake of simplicity, only the second tier is actively replicated (the problem is similar when all tiers are replicated). As shown in Figure 4, each of the second-tier replicas processes an incoming invocation from the first-tier client (1) and, in turn, invokes the third-tier server (2). There will be three identical invocations, one from each second-tier replica, issued to the third-tier; unless two of these three duplicate invocations are suppressed and not delivered, the state of the third-tier server might be corrupted by the processing of the same operation thrice (if this operation represented the withdrawal of funds from the `Account` object in our bank example, then, the account balance would be debited thrice, when it was intended to be debited only once). The third-tier object will send a reply to the second-tier replicated object (3); in turn, each second-tier replica will send an identical response to the first-tier client (4). Again, the duplicate responses from the second-tier replicas to the first-tier should be suppressed so that the state of the first-tier client is not corrupted by the processing of redundant responses. Such duplicate invocations and responses must be filtered out to guarantee consistent execution. In an asynchronous distributed system, it is difficult to guarantee the detection of duplicates at their source. Duplicate suppression should, therefore, occur at the destination as well in order to filter duplicates that escape detection at the source.

The occurrence of duplicate invocations and responses seems fairly obvious in the case of active replication. However, this problem can also manifest itself for passively replicated objects under recovery. Consider the case where the second-tier of the above example is passively replicated; assume that the primary second-tier replica fails after sending an invocation to the third-tier object, and that a new second-tier primary replica is elected. Because it might not be possible for the newly-elected primary to ascertain if the previous invocation was received, or had completed, the new primary is likely to re-issue the same invocation to the third-tier object. In this case, it is possible for the third-tier object to receive two identical invocations from the second-tier object, one before the old primary died, and another after the new primary was installed. Thus, even in the case of passive replication, mechanisms for duplicate detection and suppression are essential. It is likely, though, that duplicates are best suppressed at the destination in the case of passive replication.

Duplicate suppression rests on the existence of mechanisms to detect duplicates reliably. In FT-CORBA, this is achieved by embedding unique information about a specific request in the “service-context” field that forms a part of the standard “on-the-wire” IIOP message. Fault-tolerant ORBs are expected to generate this

context, and to pass it along as a part of any nested invocations. For this scheme to work well, the ORB hosting the request's originator (the first-tier object) must create this context in the first place, and the ORBs of all objects (the second-tier and the third-tier objects) that subsequently participate in the nested invocation must implicitly propagate this context. This requires all of the objects in the invocation chain to run on top of an FT-CORBA-aware ORB, which is yet another argument in favor of using the same FT-CORBA infrastructure within a fault-tolerant domain.

Fault-tolerant CORBA systems that do not have access to an FT-CORBA-aware ORB must resort to some other mechanism to support duplicate detection and suppression for unreplicated clients. This might take the form of a thin library underlying the unmodified, non-FT-CORBA client-side ORB; the library is equipped with mechanisms to generate, and to insert, the service-context information, just as an FT-CORBA-aware ORB would. A client-side smart proxy or a client-side portable interceptor could also achieve the same purpose. Of course, the performance overhead of accomplishing the service-context addition outside of the ORB is always greater than when an FT-CORBA-aware ORB is used.

E. Non-Determinism

A frequent assumption in building reliable CORBA systems is that each CORBA object is deterministic in behavior. This means that distinct distributed replicas of the object, when starting from the same initial state, and after receiving and processing the same set of messages in the same order, will all reach the same final state. It is this reproducible behavior of the application that lends itself so well to reliability. Unfortunately, pure deterministic behavior is rather difficult to achieve, except for very simple applications. Common sources of non-determinism include the use of local timers, operating system-specific calls, processor-specific functions, shared memory primitives, $etc.$

Non-deterministic behavior is an inevitable and challenging problem in the development of fault-tolerant systems. For active replication, determinism is crucial to maintaining the consistency of the states of the replicas of the object. Passive replication is often perceived to be the solution for non-deterministic applications. There is some truth in this perception because, with passive replication, invocations are processed only by the primary, and the primary's state is captured and then used to assign the states of the backup replicas. If the primary fails while processing an invocation, any partial execution is discarded, and the invocation is processed afresh by the new primary. Because the state updates happen only at one of the replicas, namely, at the primary replica, the results of any non-deterministic behavior of the replicated object are completely

contained, and do not wreak havoc with the replica consistency of the object.

However, there do exist situations where passive replication is not sufficient to deal with non-determinism. This is particularly true of scenarios where the non-deterministic behavior of a passively replicated object is not contained because the behavior has "leaked" to other replicated objects in the system. Consider the case where the primary replica invokes another server object based on some non-deterministic decision ($e.g.$, for load balancing, the primary replica randomly chooses one of n servers to process a credit-card transaction). If the primary replica fails after issuing the invocation, there is no guarantee that the new primary will select the same credit-card server as the old primary; thus, the system will now be in an inconsistent state because the old and the new primary replicas have communicated with different credit-card servers, both of whose states might be updated.

For passive replication to resolve non-deterministic behavior, there should be no persistent effect ($i.e.$, no lingering "leakage" of non-determinism) resulting from the partial execution of an invocation by a failed replica. This is possible if the passively replicated object does not access external components based on non-deterministic decisions/inputs, or if all accesses are performed in the context of a transaction aborted upon failure [12]. In general, though, passive replication is no cure for non-determinism.

F. Identity and Addressing

CORBA is known for its weak identity[4] and strong addressing model. Unfortunately, reliable infrastructures need a strong identity model in order to manage replicas and to maintain their consistency. Because CORBA objects can have several distinct references whose equivalence cannot be established with absolute certainty, FT-CORBA implementations need to use additional schemes for the unique identification of the replicas in each fault-tolerant domain. CORBA's location transparency also poses a problem because reliable infrastructures must often take advantage of the replicas' physical placement in several cases, $e.g.$, to elect a new primary upon the failure of an existing primary replica, to optimize distributed protocols, to manage the group membership of collocated components efficiently.

Reliable infrastructures rely on an indirect addressing scheme because the composition of the group can change over time. This requires the FT-CORBA infrastructure to maintain a custom addressing mechanism in order to map a group IOR, at run-time, onto the references of the replicas currently in the group. Information

[4]Every CORBA object can be associated with multiple distinct object reference. However, it is not possible to ascertain if any two given references "point" to the same CORBA object, simply by comparing the references. Thus, an object reference does not provide a CORBA object with a strong identity.

about the group's identity, along with the individual addresses of its constituent replicas, are encapsulated into the group IOR of the replicated object. However, an object's group IOR enumerates only the replicas that exist at the time of group IOR generation; thus, the information in the group IOR is liable to become obsolete as replicas fail and are recovered. A major challenge for any FT-CORBA infrastructure is to keep track of the current group memberships of the replicated objects that it hosts, and to update the group IORs that clients hold.

When a client invokes a method using a replicated server's group IOR, the FT-CORBA infrastructure translates this to invoke the same method on the individual server replicas whose addresses are present in the group IOR. However, if the group IOR that the client holds is stale ($i.e.$, the membership of the group has changed so that none of the object references contained in the group IOR held by the client represents a currently operational replica), then, the client will not be able to reach the replicated server even if, in fact, there exist other operational replicas that are not represented in the stale group IOR. This problem can be solved in practice by embedding into a group IOR the addresses of one or more objects with permanent or persistent addresses ($i.e.$, addresses that are guaranteed not to change over the object's lifetime); these persistent objects can act as forwarding agents that can refresh outdated client references. Typically, the FT-CORBA infrastructure's Replication Manager (shown in Figure2) acts as this forwarding persistent object. Thus, if a client tries to invoke a replica using an outdated group reference, the replica's FT-CORBA infrastructure is responsible for transparently updating the reference held by the client through standard CORBA re-direction mechanisms (such as LOCATION_FORWARD, where a recipient ORB can re-direct an incoming invocation to another address, much in the way that a post office provides forwarding services for mail).

G. Object Factories

One of the limitations of early fault-tolerant CORBA implementations was that they did not adequately support objects with methods that return object references. A classic example of this problem is illustrated by "object factories," whose sole purpose is to create and destroy objects, in response to client requests. Clients can obtain references to newly-created objects from the factory, and can subsequently invoke operations on these objects. Object factories are a very common paradigm in distributed programming, $e.g.$, in the form of the Factory design pattern [14], and many CORBA systems make extensive use of this notion. In fact, FT-CORBA makes explicit provision for a Generic Factory specifically for the purpose of instantiating replicas, as described in Section II-D. Typically, when asked to create an object, the factory instantiates the object

within its local process address space, registers the instance with its local ORB, and then returns a reference to the newly-created instance to the client that requested the object's creation. The client can subsequently use the returned reference to contact the object directly.

Thus, the FT-CORBA factory is used to instantiate replicas on specific processors. However, in the interests of fault tolerance, the object factory must itself not constitute a single point of failure. Therefore, we must consider the possibility of replicating the object factory. At the same time, we expect the replicated factory to instantiate *replicated* application objects, $i.e.$, the replicas of the object factory, independent of their own replication style, must somehow magically create a set of the same application objects, register these objects with their respective ORBs as part of a new group, and return the group IOR to the client, rather than a reference to any individual replica. Thus, although each individual factory was designed to create an individual application object, the factory replicas must be coordinated, across different processors in an asynchronous distributed system, in order to create a replicated application object.

One way of achieving this is to have the object factory code explicitly deal with group management. However, object factories are written by the application programmer, and adding group management to the object factory merely increases the complexity of the application. Furthermore, the number and identity of the new replicas must be known to the replicated factory in order for it to be able to instantiate a new group. Exposing the details of replication management to the application programmer breaks replication transparency; also, the replicas of the object factory now have two different "pieces" of state – a common state that is identical across all of the factory replicas, and an individual state that is specific to each factory replica. If the object factory is actively replicated, then, its replicas must coordinate amongst themselves to achieve the end-result of creating a replicated object and generating a group IOR. If the object factory is passively replicated, then, the backups must be equally involved in creating replicas on their respective processors; the replica creation process should not form a part of the periodic state transfer (of the common state) from the primary replica, but must occur synchronously across both the primary and the backup replicas. Thus, the backup replicas are passive w.r.t. normal operations, but are active w.r.t. the coordinated creation of a replicated object.

Another way of achieving this is to decouple the replicas of the factory from each other, and to perform the coordination of the factory replicas using a higher-level entity, such as the FT-CORBA Replication Manager (shown in Figure 2). In this case, the requests for the creation of a replicated application object are issued by clients directly to the Replication Manager, which then delegates the creation of individual application

replicas to factories on specific processors. Each factory replica creates an application replica, and returns its application replica's reference to the Replication Manager. In turn, the Replication Manager "stitches" together a group IOR using the individual application replica references that it has received from the various factories, and returns this group IOR to the client that requested the creation of the replicated object.

Regardless of whether the factories or the Replication Manager generate the group IOR, FT-CORBA provides some interfaces and mechanisms to deal with replica creation. Using the FT-CORBA Property Manager interface described in Section II-D, the application programmer must register custom factories for the various object types in the application with the fault-tolerant infrastructure. While FT-CORBA's Generic Factory interface makes the creation of replicated objects relatively straightforward, it involves significant modifications to, and requires the re-design of, existing CORBA applications.

H. Trade-Offs in Configuring Fault Tolerance

The FT-CORBA specification permits considerable latitude in terms of configuring fault tolerance to suit an application's requirements. This is possible through the various fault tolerance properties that can be assigned values by the user at the time of deploying an FT-CORBA application. With this flexibility also comes the potential for abuse – selecting the wrong replication style for a specific object might adversely impact its performance, selecting the wrong fault detection timeout for an object might lead to its being suspected as having failed far too often, $etc.$

Investing the effort to consider the various trade-offs ($e.g.$, active replication $vs.$ passive replication) in a resource-aware manner will allow FT-CORBA infrastructures to work efficiently, to make the best possible use of the available resources, and to provide fault tolerance with superior performance. One of the most important sets of trade-offs occurs in choosing between the active and passive replication styles for a replicated object:

- **Checkpointing.** With cold passive replication, under normal operation, the primary replica's state is checkpointed into a log. If the state of the object is large, this checkpointing could become quite expensive. With warm passive replication, if the state of the object is large, transferring the primary's state to the backup replicas, even if it is done periodically, could become quite expensive. This state transfer cost is incurred for active replication only when a new active replica is launched, and never during normal operation.
- **Computation.** Cold passive replication requires only one replica to be operational and, thus, consumes CPU cycles only on one processor. While warm passive replication requires more replicas to be operational,

these backups do not perform any operations (other than receiving the primary replica's state periodically), and also conserve CPU cycles on their respective processors. With active replication, every replica performs every operation, and therefore consumes the same number of CPU cycles of its respective processor. Thus, passive replication consumes cycles on fewer processors during normal operation, $i.e.$, it does not require normal fault-free operations to be performed by each of the replicas on its respective processor. For operations that are compute-bound, $i.e.$, require many CPU cycles, the cost of passive replication can be lower (in the fault-free case) than that of active replication.

• **Bandwidth usage.** For active replication, a multicast message is required to issue the operation to each replica. This can lead to increased usage of network bandwidth because each operation may itself generate further nested operations. For passive replication, because only one replica, the primary client (server) replica, invokes (responds to) every operation, less bandwidth may be consumed. However, if the state of the primary replica is large, the periodic state transfer may also require significant network bandwidth.

• **Speed of recovery.** With active replication, recovery time is faster in the event that a replica fails. In fact, because all of the replicas of an actively replicated object perform every operation, even if one of the replicas fails, the other operational replicas can continue processing and perform the operation. With passive replication, if the primary replica fails, recovery time may be significant. Recovery in passive replication typically requires the re-election of a new primary, the transfer of the last checkpoint, and the application of all of the invocations that the old primary received since its last checkpoint. If the state of the object is large, retrieving the checkpoint from the log may be time-consuming. Warm passive replication yields faster recovery than cold passive replication.

The cost of using active $vs.$ passive replication is also dictated by other issues, such as the number of replicas and the depth of nesting of operations. For a CORBA object, active replication is favored if the cost of network bandwidth usage and the cost of CPU cycles is less than the cost incurred in passive replication due to the periodic checkpointing of the object's state.

Hybrid active-passive replication schemes [16] have been considered, with the aim of addressing the reduction of multicast overhead in active replication styles, as well as of achieving the best of the active and passive replication styles. An approach for marrying both replication techniques has also been proposed in [9]. At the protocol level, this system uses a variant of a distributed consensus protocol that acts as a common denominator between both replication styles. An important property of this system is that both active and passive replication techniques can be used at the same time in a distributed application, and a unique feature is that

the replication technique can be dynamically specified on a per-operation basis.

I. Common Limitations

Regardless of the specific approach (interception, integration, service, or FT-CORBA) used, the following holds true of current fault-tolerant CORBA systems:

- Whenever a reliable ordered group communication toolkit is employed to convey the messages of the CORBA application, the resulting fault-tolerant CORBA system will require the group communication toolkit to be ported to new operating systems, as required.
- A CORBA object is associated with application-level state, ORB/POA -level state and infrastructure-level state; for effective fault tolerance and strong replica consistency, all three kinds of state must be maintained identical across all replicas of the same object. Even if a CORBA object is stateless w.r.t. application-level state, the other two kinds of state nevertheless exist.
- As long as a CORBA object has application-level state, true server-side transparency ($i.e.$, no modifications to the server code) cannot be fully achieved in a portable manner.
- Although a CORBA object is widely regarded as the unit of replication, the process containing the CORBA object is, for all practical purposes, the correct unit of replication due to the presence of unavoidable in-process state.
- Replicas of a CORBA object cannot currently be supported across different ORB implementations while preserving strong replica consistency, $e.g.$, it is not possible for a CORBA object to have one replica using VisiBroker and the other using Orbix.
- Replicas of a CORBA object cannot currently be supported across different FT-CORBA implementations, even if the same ORB is used by all of the replicas, $e.g.$, it is not possible for a CORBA object to have one replica supported by Eternal and the other supported by OGS.
- The CORBA application must be deterministic in behavior so that, when replicas are created and distributed across different processors, the states of the replicas will be consistent, as the replicas process invocations and responses, and even if faults occur in the system.
- If an unreplicated client (server) that is not supported by a fault-tolerant CORBA infrastructure communicates with a replicated server (client), replica consistency might be violated if a fault occurs, even if gateways are used.
- There is no support for the consistent remerging of the replicas of CORBA objects following a network

partition (most of the approaches assume a primary partition model, which allows only one component, called the primary, to continue operating, while the other disconnected components cease to operate).

- Design faults, $i.e.$, intrinsic problems in the application that cause all of the replicas of an object to crash in the same way, are not tolerated. Current fault-tolerant CORBA systems do not use software fault-tolerance mechanisms such as design diversity or N-version programming [2] in order to remedy this deficiency.
- Faults are assumed to be independent, $i.e.$, processors, processes and objects fail independently of each other (also known as the *independent-failures assumption*). Thus, correlated, or common-mode, failures are not handled.

IV. Conclusion

The emergence of object-oriented middleware such as CORBA has greatly simplified the development of distributed applications, by allowing programmers to build systems from components that interact seamlessly across multiple processors, transcending differences in programming languages, physical locations, operating systems, byte orders, hardware architectures, $etc.$ As applications become more distributed and complex, the likelihood of faults undoubtedly increases because individual processors and communication links can fail independently. For almost a decade since its inception, CORBA had no standard support for fault-tolerance. Various research efforts were expended to remedy this deficiency, each of the resulting systems using replication to protect applications from the failure of individual objects. The recently adopted specification for Fault-Tolerant CORBA (FT-CORBA) represents the marriage of several of these efforts and their insights, and comprises the specifications necessary to replicate CORBA objects.

The lessons that we have learned as implementors of reliable CORBA systems and as contributors to the FT-CORBA standard have been captured in this paper in the form of recommendations and cautions to CORBA application programmers, ORB vendors, FT-CORBA developers, and potential users of the new FT-CORBA standard or of existing fault-tolerant CORBA infrastructures. While some of the insights that we describe might seem rather intuitive in hindsight, our experience has shown us that these practices are often sadly neglected in the development of reliable distributed applications, and that these lessons are learned the hard, and often costly, way.

We strongly believe that *reliability should not be an after-thought*. Fault tolerance can be added transparently only to very simple applications. Real-world applications can use many proprietary mechanisms, can communicate with legacy systems, can work with commercial databases, and can exhibit non-deterministic

behavior. In such cases, it should not be assumed that the use of an FT-CORBA infrastructure "out-of-the-box" will provide a ready solution for complicated applications. FT-CORBA cannot magically resolve non-determinism in CORBA applications; for example, if multithreading is used by the application, then, the application programmer must take care to ensure that threads do not update shared in-process state concurrently.

For real-world complex applications, there might be significant re-architecturing of the application if reliability is an after-thought. Investing the thought and the effort to plan ahead for reliability during the design of a new application can save the cost of re-design and re-writing of the application when fault tolerance does become an issue. Planning ahead might involve examining (i) the important system/application state, $i.e.$, the data that will need to be protected despite faults in the system, (ii) the appropriate granularity of the application's objects (because replicating many small objects might impact performance or resources), (iii) the critical elements of processing, $i.e.$, the processing or operations that will need to continue uninterrupted, despite faults in the system, and (iv) the data flows within the system (because objects that communicate frequently might need to be collocated within the same process).

Unfortunately, the FT-CORBA specification, while being rather detailed, does have some practical limitations, and does not fully address some of the issues faced by developers of real-world CORBA applications. The problems of providing fault tolerance for the complex and critical CORBA applications of the future are far from over. FT-CORBA cannot fix problems that already exist in applications and, therefore, does not provide solutions for cases where applications may crash due to design errors (with FT-CORBA, if one replica dies due to a divide-by-zero exception, all of the replicas will die identically). Furthermore, if the system partitions so that some of the replicas are disconnected from other replicas of the same object, FT-CORBA infrastructures cannot automatically reconcile any differences in the states of the replicas when communication is once more re-established. Other open issues, such as dealing with new sources of non-determinism, supporting the CORBA component model, combining fault tolerance and real-time, combining fault tolerance and security, and combining replication and transactions, are all practical challenges that will undoubtedly be faced by users, and will need to be addressed by the developers of reliable systems.

In this paper, we have shared our experiences and insights as users of multiple CORBA implementations, developers of reliable CORBA infrastructures and fault-tolerant CORBA applications, and as contributors to the FT-CORBA standardization process. We have discussed the challenges that are commonly faced in developing fault-tolerant CORBA implementations, the pitfalls encountered when building reliable applications,

and how best to take advantage of the FT-CORBA standard. It is our sincere hope that FT-CORBA users and developers alike will benefit from our knowledge, experiences and contributions, and that our insights will help to shape the future of fault-tolerant infrastructures for middleware and distributed object applications.

Acknowledgments. We would like to acknowledge the anonymous reviewers whose detailed comments and feedback greatly improved the layout, the content and the quality of this paper.

REFERENCES

[1] K. P. Birman and R. van Rennesse. *Reliable Distributed Computing Using the Isis Toolkit*. IEEE Computer Society Press, 1994.

[2] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proceedings of the Fault-Tolerant Computing Symposium*, pages 3–9, Toulouse, France, July 1978.

[3] D. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[4] D. H. Craft. A study of pickling. *Journal of Object-Oriented Programming*, 5(8):54–66, 1993.

[5] M. Cukier, J. Ren, C. Sabnis, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. Schantz. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the IEEE 17th Symposium on Reliable Distributed Systems*, pages 245–253, West Lafayette, IN, October 1998.

[6] J. C. Fabre and T. Perennou. A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers*, 47(1):78–95, 1998.

[7] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1998.

[8] P. Felber. Lightweight fault tolerance in CORBA. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'01)*, pages 239–247, Rome, Italy, September 2001.

[9] P. Felber, X. Défago, P. Eugster, and A. Schiper. Replicating CORBA objects: A marriage between active and passive replication. In *Proceedings of the 2nd IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*, pages 375–387, Helsinki, Finland, June 1999.

[10] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[11] P. Felber, B. Jai, R. Rastogi, and M. Smith. Using semantic knowledge of distributed objects to increase reliability and availability. In *Proceedings of the 6th International Workshop on Object-oriented Real-time Dependable Systems (WORDS'01)*, pages 153–160, Rome, Italy, January 2001.

[12] P. Felber and P. Narasimhan. Reconciling Replication and Transactions for the End-to-End Reliability of CORBA Applications. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'02)*, pages 737–754, Irvine, California, October 2002.

[13] R. Friedman and E. Hadad. FTS: A high performance CORBA fault tolerance service. In *Proceedings of IEEE Workshop on Object-oriented Real-time Dependable Systems*, San Diego, CA, January 2002.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[15] R. Guerraoui, P. Felber, B. Garbinato, and K. Mazouni. System support for object groups. In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, October 1998.

[16] H. Higaki and T. Soneoka. Fault-tolerant object by group-to-group communiations in distributed systems. In *Proceedings of the Second International Workshop on Responsive Computer Systems*, pages 62–71, Saitama, Japan, October 1992.

[17] IONA and Isis. *An Introduction to Orbix+Isis*. IONA Technologies Ltd. and Isis Distributed Systems, Inc., 1994.

[18] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, University of Zurich, February 1995.

[19] C. Marchetti, M. Mecella, A. Virgillito, and R. Baldoni. An interoperable replication logic for CORBA systems. In *Proceedings of the 2nd International Symposium on Distributed Objects and Applications (DOA'00)*, pages 7–16, February 2000.

[20] G. Morgan, S. Shrivastava, P. Ezhilchelvan, and M. Little. Design and implementation of a CORBA fault-tolerant object group service. In *Proceedings of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems*, Helsinki, Finland, June 1999.

[21] L. Moser, P. Melliar-Smith, and P. Narasimhan. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems*, 4(2):81–92, 1998.

[22] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[23] S. Mullender, editor. *Distributed Systems*, chapter 7 and 8. Addison-Wesley, 2nd edition, 1993.

[24] P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, December 1999.

[25] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Providing support for survivable CORBA applications with the Immune system. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 507–516, Austin, TX, May 1999.

[26] P. Narasimhan, L. Moser, and P. M. Melliar-Smith. Gateways for accessing fault tolerance domains. In *Proceedings of Middleware 2000*, New York, USA, April 2000.

[27] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. Doors: Towards high-performance fault tolerant CORBA. In *Proceedings of the 2nd International Symposium on Distributed Objects and Applications (DOA'00)*, pages 39–48, February 2000.

[28] Object Management Group. The Common Object Services specification. OMG Technical Committee Document formal/98-07-05, July 1998.

[29] Object Management Group. Fault Tolerant CORBA (final adopted specification). OMG Technical Committee Document formal/01-12-29, December 2001.

[30] Object Management Group. Portable Interceptors (final adopted specification). OMG Technical Committee Document formal/01-12-25, December 2001.

[31] Object Management Group. The Common Object Request Broker: Architecture and specification, 2.6 edition. OMG Technical Committee Document formal/02-01-02, January 2002.

[32] G. Parrington, S. Shrivastava, S. Wheater, and M. Little. The design and implementation of Arjuna. *USENIX Computing Systems Journal*, 8(3):255–308, Summer 1995.

[33] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[34] B. S. Sabnis. Proteus: A software infrastructure providing dependability for CORBA applications. Master's thesis, University of Illinois at Urbana-Champaign, 1998.

[35] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. *Software - Practice and Experience*, 28(9):963–79, July 1998.

[36] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[37] A. Vaysburd and K. Birman. The Maestro approach to building reliable interoperable distributed applications with multiple execution styles. *Theory and Practice of Object Systems*, 4(2):73–80, 1998.

[38] W. Vogels, R. V. Renesse, and K. Birman. Six misconceptions about reliable distributed computing. In *Proceedings of the 8th ACM SIGOPS European Workshop*, September 1998.

[39] Y. M. Wang, Y. Huang, K. P. Vo, P. Y. Chung, and C. M. R. Kintala. Checkpointing and its applications. In *Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing*, pages 22–31, Pasadena, CA, June 1995.

APPENDIX

I. CODE LISTINGS

```
1  interface Account {
2    readonly attribute long number;
3    readonly attribute string owner;
4    readonly attribute float balance;
5
6    void deposit(in float amount);
7    void withdraw(in float amount);
8  };
```

Listing 1: IDL interface of the bank `Account` object.

```
1  int main(int argc, char *argv[])
2  {
3    // Initialize the ORB
4    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
5
6    // Obtain the address of the Account object from the command-line
7    Object_var ior = orb->string_to_object(argv[1]);
8    Account_var acc = Account::_narrow(ior);
9
10   // Invoke server
11   acc->deposit(100);
12   cout << "New balance is " << acc->balance() << endl;
13 }
```

Listing 2: C++ implementation of the bank client.

```
1  class Account_impl : POA_Account
2  {
3    CORBA::Long number_;
4    string owner_;
5    CORBA::Float balance_;
6
7  public:
8    Account_impl(CORBA::Long n, string o) :
9      number_(n), owner_(o), balance_(0.0) {}
10
11   // Account IDL operations
12   CORBA::Long number()
13     { return number_; }
14   char* owner()
15     { return CORBA::string_dup(owner_.c_str()); }
16   CORBA::Float balance()
17     { return balance_; }
18   CORBA::Float deposit(CORBA::Float amount)
19     { balance_ += amount; }
20   CORBA::Float withdraw(CORBA::Float amount)
21     { if(amount > balance_) balance_ -= amount; }
22 };
```

Listing 3: C++ implementation of the bank `Account` object.

```
1  // IDL
2  interface Account : FT::Updateable {
3    // Same as before ...
4  };
5
6  // C++
7  class Account_impl : POA_Account
8  {
9    // Some as before ...
10 public:
11   // Checkpointable (base class of Updatable) operations : state transfer
12   FT::State* get_state(); // Code omitted
13   set_state(const FT::State& s); // Code omitted
14
15   // Updatable operations : state update
16   FT::State* get_update(); // Code omitted
17   set_state(const FT::State& s); // Code omitted
18 };
```

Listing 4: IDL interface and C++ implementation of the replicated bank `Account` object.