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

Pascal Felber, *Member*, *IEEE*, and Priya Narasimhan, *Member*, *IEEE*

Abstract—It has been almost a decade since the earliest reliable CORBA implementation and, despite the adoption of the Fault-Tolerant CORBA (FT-CORBA) standard by the Object Management Group, CORBA is still not considered the preferred platform for building dependable 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 deploying reliable CORBA applications, and the fact that current FT-CORBA do not lend themselves readily to complex, real-world applications. In this paper, we candidly share our independent experiences as developers of two distinct reliable CORBA infrastructures (OGS and Eternal) and as contributors to the FT-CORBA standardization process. Our objective is to reveal the intricacies, challenges, and strategies in developing fault-tolerant CORBA systems, including our own. Starting with an overview of the new FT-CORBA standard, we discuss its limitations, along with techniques for best exploiting it. We reflect on the difficulties that we have encountered in building dependable CORBA systems, the solutions that we developed to address these challenges, and the lessons that we learned. Finally, we highlight some of the open issues, such as nondeterminism and partitioning, that remain to be resolved.

Index Terms—CORBA, FT-CORBA, fault tolerance, nondeterminism, replication, recovery, OGS, Eternal.

1 Introduction

The integration of distributed computing with object-oriented programming has led to distributed object middleware, where objects are distributed across processors. Typical middleware applications consist of client objects invoking operations on, and receiving responses from, remote server objects, through messages sent across the network. The Common Object Request Broker Architecture (CORBA) [31] is a standard for middleware that was established by the Object Management Group.

CORBA uses its Interface Definition Language (IDL) to define interfaces to server objects. CORBA's language transparency implies that a client needs to be aware of only the IDL interface, and not the language-specific implementation, of its server object. CORBA's interoperability enables clients and servers to communicate over the TCP/IP-based Internet Inter-ORB Protocol (IIOP), despite heterogeneity in their respective platforms and operating systems. CORBA's location transparency allows clients to 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 the recently adopted Fault-Tolerant CORBA standard (FT-CORBA) [29], CORBA had no intrinsic support for reliability. Early fault-tolerant CORBA systems (which preceded the FT-CORBA standard) adopted diverse approaches:

- The *integration approach*, where support for replication is integrated into the ORB (e.g., systems such as Electra [17], Orbix+Isis [16], and Maestro [37]),
- The *interception approach*, where support for replication is provided transparently underneath the ORB (e.g., Eternal [23]), and
- The *service approach*, where support for replication is provided primarily through a collection of CORBA objects above the ORB (e.g., systems such as OGS [6], AQuA [4], DOORS [27], Newtop [19], FRIENDS [5], FTS [12], and IRL [18]).

Many of these systems have contributed to our collective understanding of distributed fault tolerance for CORBA and have, directly or indirectly, influenced the FT-CORBA standard.

However, there exist some inherent drawbacks in these systems (e.g., regular clients cannot interact with replicated servers in a fault-tolerant manner); furthermore, each system exposes different APIs to the application, making the development of portable fault-tolerant CORBA applications almost impossible. While the FT-CORBA standard aims to resolve these issues through standardized CORBA APIs for reliability, it fails to address the nontrivial challenges (e.g., nondeterminism) in providing fault tolerance to real-world CORBA applications.

This paper focuses on 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 can be gleaned, as a result. Our wealth of experience—both as independent implementors of distinct fault-tolerant CORBA implementations (Eternal

- *P. Felber is with Institut Eurecom, 2229 Routes des Cretes, BP 193, 06904 Sophia Antipolis, France. E-mail: pascal.felber@eurecom.fr.*
- *P. Narasimhan is with the Electrical and Computer Engineering Department, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213. E-mail: priya@cs.cmu.edu.*

Manuscript received 5 Dec. 2002; revised 28 Mar. 2003; accepted 11 Aug. 2003.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 117885.

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 for real-world deployment.

1.1 Replication

Despite their differences, the various approaches to fault-tolerant CORBA are alike in their use of replication to protect the application against faults. 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.

1.1.1 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) 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, nonduplicate, invocation (response).

The integration, interception, and service approaches to fault-tolerant CORBA all 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, should 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 loss of data, messages or consistency.

1.1.2 Replication Styles

There are essentially two kinds of replication styles—active replication and passive replication [22]. With *active* (also known as state-machine) 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 existing 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.

1.1.3 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 by hiding the number, the identities, and the locations of the replicas from other objects in the system. With an object group being equivalent to a replicated CORBA object, group communication[1] 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, several of the various fault-tolerant CORBA systems described in this paper employ totally ordered reliable multicast group communication toolkits to facilitate consistent replication.

1.1.4 Relevant Non-CORBA Systems

This discussion of replication would not be complete without a brief overview of reliable distributed 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 the science and engineering behind distributed fault tolerance. It is no

1. Group communication systems treat a set of processes or objects as a logical group and provide primitives for sending messages simultaneously to the group as a whole, usually with various ordering guarantees on the delivered messages.

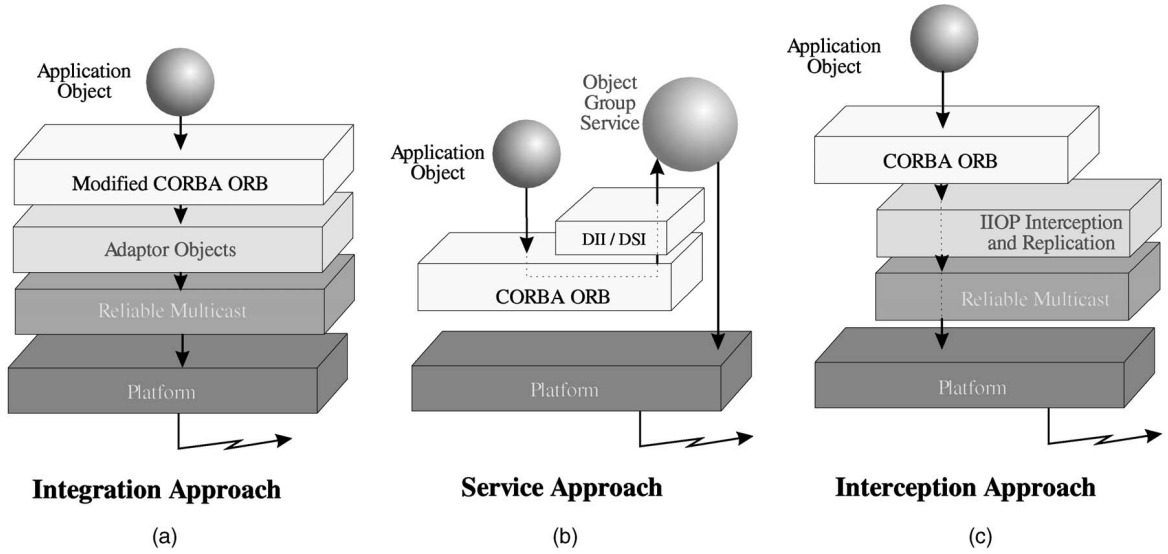

Fig. 1. Different approaches to fault-tolerant CORBA.

surprise that almost every fault-tolerant CORBA system embodies principles that are derived from one or the other of these systems.

2 Existing Fault-Tolerant CORBA Systems

Initial efforts to enhance CORBA with fault tolerance leaned toward 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 underlying system model for all three approaches is an asynchronous distributed system, where processors communicate via messages over a network that is completely connected, i.e., network partitioning does not occur. Communication channels are not assumed to be FIFO or authenticated and the system is asynchronous in that no bounds can be placed on computation times or message-transmission latencies. Processors have access to local clocks that are not necessarily synchronized across the system. The fault model includes 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 reinstatement to correct operation.

2.1 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 addition of group communication support directly into the ORB is likely to involve replacing CORBA's TCP/IP-based IIOP transport by a proprietary group communication protocol. The resulting modified, but fault-tolerant, ORB may therefore be noncompliant 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 and that the details of the replica consistency mechanisms (buried within the ORB) can be hidden from the application programmer.

Electra: Developed at the University of Zurich, Electra [17] 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 Fig. 1a, 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 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 [16] was the first commercial offering in the way of fault tolerance support for CORBA applications. Like Electra, Orbix+Isis modifies the internals of the ORB to accommodate the use of the Isis toolkit [1] for the reliable ordered multicast of messages.

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. 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.

Maestro Replicated Updates ORB: Developed at Cornell University, Maestro [37] is CORBA-like middleware 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. In particular, the Replicated Updates request manager supports the active replication of server objects. "Smart" clients have access to compound Interoperable Object References[2] (IORs) consisting of the enumeration of the addresses of all of the replicas of a server object. The client connects to a single server replica; if this replica fails, the client reconnects 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 normal 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 Maestro's request managers 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 can be used to add reliability and high availability to CORBA applications where it is not feasible to make modifications at the client side.

2.2 The Service Approach

The service approach to enhancing CORBA involves adding a new service, along the lines of CORBA's existing Common Object Services [28]. Because a CORBA service is 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 a CORBA service, the CORBA application needs to be explicitly aware of the service objects; the application code is, therefore, likely to require modification.

Using this approach, fault tolerance can be provided as a CORBA service. Of course, because the objects that provide reliability reside above the ORB, every interaction with these service objects necessarily passes through the ORB and incurs the associated performance overheads.

2. An Interoperable Object Reference (IOR) is a string-like reference to a CORBA object and contains one or more profiles. Each profile has enough information—typically, the host name, port number, and object key—for the client ORB to contact the object using some protocol, usually IIOP.

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 developer to select the replication style (cold 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. For each object in the system, the ReplicaManager maintains a repository that stores the number of replicas, the hosts on which the replicas are running, the status of each replica, and the number of faults seen by the replica on a given host. As part of the state of the ReplicaManager, this repository 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) [6], [9] 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 comprises a number of subservices implemented on top of off-the-shelf CORBA ORBs. The multicast subservice provides for the reliable unordered multicast of messages to server replicas; the messaging subservice provides the low-level mechanisms for mapping these messages onto the transport layer; the consensus subservice imposes a total order on the multicast messages; the membership subservice keeps track of the composition of obect groups; finally, the monitoring subservice detects crashed objects. Each of these IDL-specified subservices is independent and is itself implemented as a collection of CORBA objects.

To exploit the facilities of the OGS objects, the server objects of the application need to be modified. The server replicas inherit a common IDL interface that permits them to join or leave the group of server replicas and allows OGS to perform state transfer actions.

OGS provides a client object with a local proxy for each replicated server with which the client communicates. The client-side server's proxy and the server-side OGS objects are together responsible for mapping client requests and server responses onto multicast messages that convey the client-server communication. The client establishes communication with a replicated server object by binding to an identifier that designates the object group representing all of the server replicas. Using this object group identifier, the client can then direct its requests to the replicated server object; once the client is bound to the server's object group, it can invoke the replicated server as if it were invoking a

single unreplicated server. However, because the client is aware of the existence of the server replicas and can even obtain information about the server object group, the server's replication is not necessarily transparent to the client. Clients need to be modified to bind, and to dispatch requests, to a replicated server.

Newtop Object Group Service: Developed at the University of Newcastle, the Newtop [19] service provides fault tolerance to CORBA using the service approach. While the fundamental ideas are similar to OGS, 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 subgroups, with each subgroup being connected within itself. Total ordering continues to be preserved within each subgroup. However, no mechanisms are provided to ensure consistent remerging once communication is reestablished between the subgroups.

IRL and FTS: The Interoperable Replication Logic (IRL) [18] and FTS [12] are two different systems that provide fault tolerance to CORBA applications through a service approach. Both IRL and FTS were developed after the adoption of the FT-CORBA standard, and aim to provide the client-side replication support required by the new FT-CORBA standard (see Section 2.4).

IRL aims to uphold CORBA's interoperability by supporting fault-tolerant CORBA applications that are composed of objects running over implementations of ORBs from different vendors. FTS aims to provide some support for network partitioning by imposing a primary component model (where operation is sustained in one of the components, known as the primary) if the system partitions into disconnected components.

AQuA: Developed jointly by the University of Illinois at Urbana-Champaign and BBN Technologies, AQuA [4] is a framework for building fault-tolerant CORBA applications. AQuA employs the Ensemble/Maestro [35], [37] toolkits and is comprised of 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 kinds of faults to tolerate, the replication policy, the degree of replication, the type of voting, 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 to detect timing faults and voters which either accept the first invocation/response or perform majority voting on the received invocations/responses.

AQuA provides mechanisms for application-level majority voting 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 [5] system aims to provide mechanisms for building fault-tolerant applications through the use of libraries of meta-objects 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.

2.3 The Interception Approach

The interception approach to extending CORBA with new functionality involves providing fault tolerance transparently through an interceptor, a software component that can attach itself to existing precompiled and prelinked binaries of applications. The interceptor can contain additional code to modify the application's behavior, without the application or the ORB being aware of the interceptor's existence or operation.

However, if the interception mechanisms are specific to the operating system, as is often the case, then the interceptor needs to be ported to every operating system that is intended to run the CORBA application. CORBA's Portable Interceptors [30] aim at providing standardized interception "hooks" within the ORB so that this porting effort is eliminated.

Eternal: Developed at the University of California, Santa Barbara, the Eternal system [20], [23] exploits the interception approach to provide fault tolerance to CORBA applications. The mechanisms implemented in different parts of the Eternal system work together to provide strong replica consistency without requiring the modification of either the application or the ORB; this allows Eternal to work with both C++ and Java ORBs, currently including VisiBroker, Orbix, Orbacus, ILU, TAO, e*ORB, omniORB2, and CORBAplus.

Eternal conveys the application's IIOP messages over the reliable totally ordered multicast messages of the underlying Totem system [21]. Eternal's 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. Different replication styles—active, cold passive, warm passive, and hot passive replication—of both client and server objects are supported. Eternal's 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 Eternal's Replication Mechanisms for multicasting via Totem. Eternal's Replication Mechanisms, together with its 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 nondeterminism that multithreaded CORBA applications exhibit.

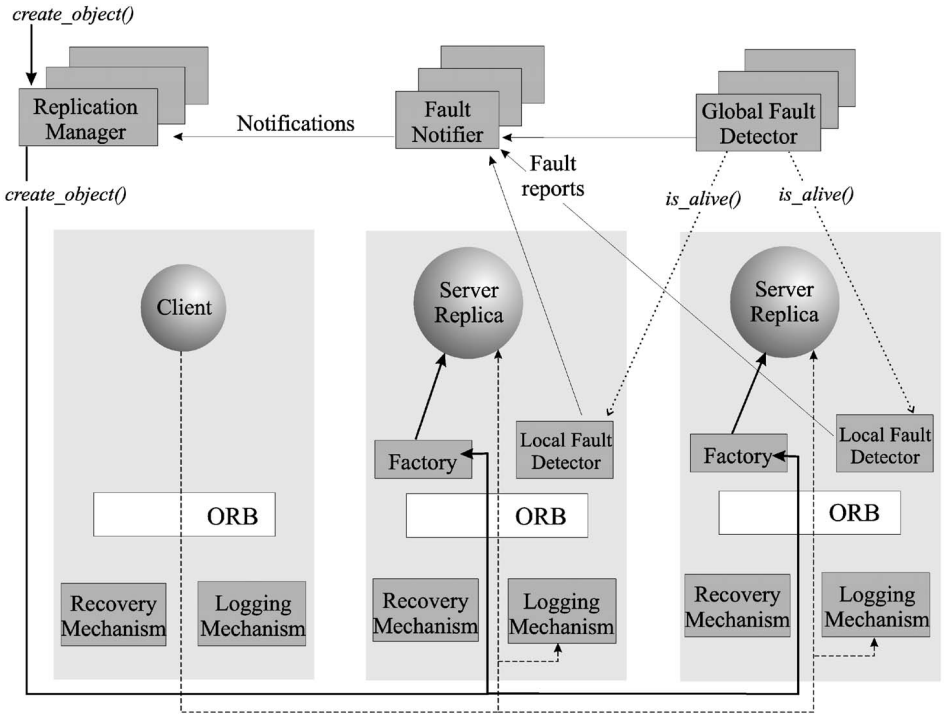

Fig. 2. Architectural overview of the Fault-Tolerant CORBA (FT-CORBA) standard.

Eternal tolerates communication faults, including message loss and network partitioning, and processor, process, and object faults. To tolerate value faults in the application, Eternal employs more stringent protocols to support active replication with majority voting [25] on both the invocations and the responses for every application object.

2.4 The Fault-Tolerant CORBA Standard

The recent Fault-Tolerant CORBA (FT-CORBA) standard describes minimal fault-tolerant mechanisms to be included in any CORBA implementation, as well as interfaces for supporting more advanced fault tolerance. FT-CORBA implementors are free to use proprietary mechanisms (such as reliable multicast protocols) for their actual implementations as long as the resulting system complies with the specified interfaces and the behavior expected of those interfaces.

Because some developers of fault-tolerant CORBA systems (namely, [20], [27], and [23]) were involved in the standardization, FT-CORBA builds on the experiences from these systems. 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 general, but complex, framework. For instance, FT-CORBA can be implemented through different mechanisms, such as group communication and replicated databases. The client-side mechanisms that CORBA implementations must support—regardless of whether they implement the server-side FT-CORBA mechanisms—are deliberately minimal. The client-side mechanisms support multiprofile object references, where each profile designates a server replica, along with rules for iterating through the profiles, if a replica fails. This allows unreplicated clients to interact with replicated FT-CORBA-supported servers in a fault-tolerant manner. At the time of writing, Eternal [20] and DOORS [27] support both client-side and server-side FT-CORBA mechanisms; IRL [18] and FTS [12] include support for FT-CORBA's client-side mechanisms.

Fig. 2 shows the architecture of the FT-CORBA specification. The Replication Manager replicates objects and distributes the replicas across the system. Although each server replica has an individual reference, the Replication Manager fabricates an object-group reference that clients can use to contact the replicated server. 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 configure each object's reliability through fault-tolerance properties, such as the *Replication Style* (stateless, active, cold passive, or warm passive replication), the *Membership Style* (whether the addition, or removal, of replicas is application-controlled or infrastructure-controlled), the *Consistency Style* (whether recovery, checkpointing, logging, etc., are application-controlled or infrastructure-controlled), list of factories (objects that can create and delete replicas), the *Initial Number of Replicas*, the *Minimum Number of Replicas* to be maintained, the *Checkpoint Interval* (time interval between successive checkpoints of the object's state), the *Fault Monitoring Style* (whether the object is periodically "pinged" or instead sends periodic "i-am-alive" messages), the *Fault Monitoring Granularity* (whether the object is monitored on the basis of an individual replica, a location, or a location-and-type), and the *Fault Monitoring Interval* (the frequency at which an object is to be "pinged" for fault-detection).

The Generic Factory allows users to create replicated objects in the same way that they would create unreplicated

TABLE 1
Comparison of Different Approaches to Fault-Tolerant CORBA

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, FTS, DOORS, FRIENDS, Newtop, AQuA, IRL
Interception	Transparent to the application and the ORB 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	Leaves implementation details open Requires extensions to the standard ORB core	Eternal, DOORS

objects. The Object Group Manager allows users to directly control 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 CORBA 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 processor. On receiving reports of faults from the Fault Detector, the Fault Notifier filters them to eliminate duplicate reports. The Fault Notifier then distributes fault reports to all interested parties. The Replication Manager, being a subscriber of the Fault Notifier, can initiate appropriate recovery actions on receiving fault reports.

One limitation of the FT-CORBA specification is that it requires all of the replicas of an object to be deployed on the same FT-CORBA infrastructure, i.e., heterogeneity is not supported within a group. Also, FT-CORBA requires CORBA applications to be deterministic and does not protect them against partitioning faults. A comparison of differenct approaches to FT-CORBA is shown in Table 1.

3 Critical Look at Fault-Tolerant CORBA Systems

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.

3.1 Server-Side and Client-Side Transparency

Replication transparency hides the use of replication from the application programmer by providing the illusion that messages are sent to, and received from, single objects. This transparency is clearly advantageous to the application programmer; not only does it preserve the current CORBA programming model, but it also relieves the application programmer from having to deal explicitly with the difficult issues of fault tolerance.

With client-side transparency, the client is unaware of the server's replication and its code does not need to be modified to communicate with the replicated server. The main difficulty in achieving client-side transparency is to make the replicated server's group reference (i.e., reference containing the addresses of all of the server replicas) appear the same as the unreplicated server's object 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 proprietary reliable ORBs. The interception approach (Eternal) transparently maps CORBA's object references (IORs) to custom group references that are hidden from the client application. The service approach (OGS and IRL) lets 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; 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 ummodified client application code can generally be used 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. With server-side transparency, 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

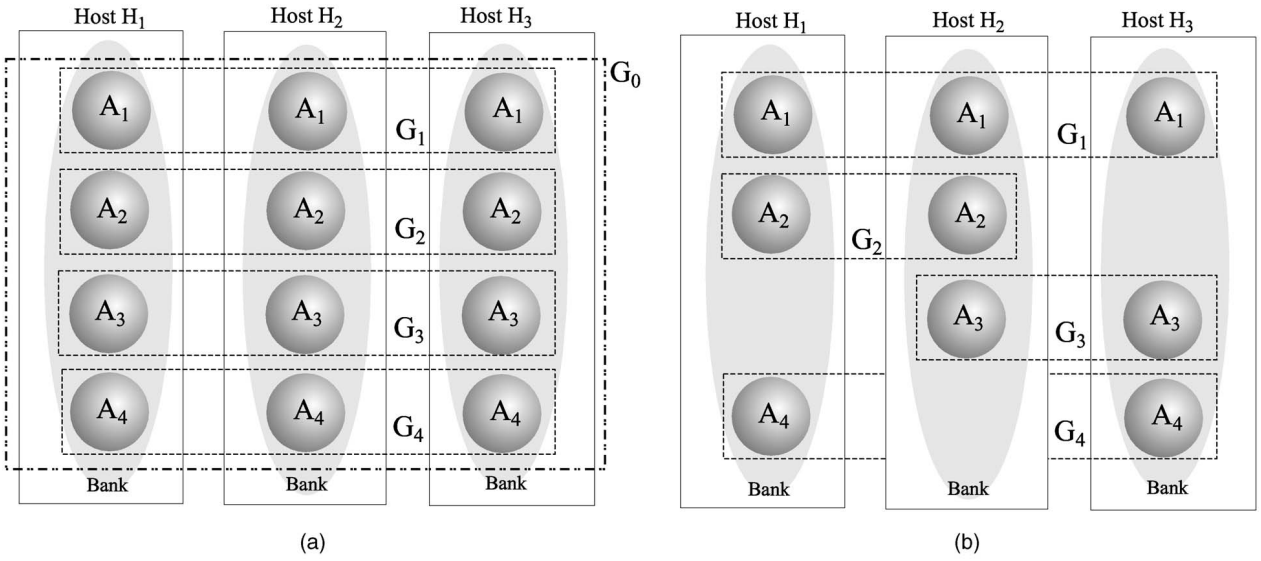

Fig. 3. Processors hosting replicas of an object A in two configurations: (a) fully overlapping groups and (b) partially overlapping groups.

servers. Server-side transparency is harder to achieve than client-side replication because of the existence of server state. For the purposes of recovery and consistent replication, there needs to exist some way of retrieving and transferring the state of a server replica. Because an object's state consists of the values of its application-level data structures, there must be some provision for extracting this state. FT-CORBA requires every replicated object to inherit the `Checkpointable` (or `Updateable`) interface that defines operations for the retrieval and the assignment of the complete (incremental) state of the object. The complete state refers to the entire state of the server object, while the partial state (known as an update or a state increment) refers to only that part of the object's state that has been modified since the last state snapshot. *Every* fault-tolerant CORBA system that supports stateful servers with consistent replication requires each object to support such interfaces for state transfer.

Thus, true server-side transparency (i.e., absolutely no modifications to the server code) is impossible to accomplish so long as the server must support these additional interfaces for state transfer. Of course, server-side transparency is achievable for stateless CORBA objects or if there exist other ways of manipulating/accessing an object's state other than through an IDL interface, e.g., through a pickling mechanism [3] for taking snapshots of the operating system's layout of a process' state. However, for all practical purposes, no fault-tolerant CORBA infrastructure ever fully achieves server-side transparency.

Both CORBA and FT-CORBA are server-centric and support only the notions of handling server state through server interfaces. Objects that play both client and server roles (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. For recovering such objects, the `Checkpointable` and `Updateable` interfaces need to handle both the client-side and the server-side state. Other issues with dual-role objects are discussed further in Section 3.4.

Finally, it has been argued that, even when transparency is technically achievable, it leads to generic protocols that are conservative and that cannot perform application-specific optimizations, thereby resulting in poor performance [38]. One solution [10] involves using semantic application information (e.g., knowledge that operations are read-only, deterministic, or commutative) to determine the optimal protocols for replica consistency.

3.2 Object versus Processes

Group communication toolkits have traditionally dealt with process groups; with FT-CORBA, the fundamental unit of replication is the CORBA object and groups must deal with objects rather than processes. A CORBA application containing a large number of objects might, therefore, lead to the creation and management of a large corresponding number of object groups; this is known as the group proliferation problem [14].

In particular, if any one of the processors fails, all of the 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 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 Fig. 3b. This is often the case when replicas are created and hosted across a subset of a pool of processors.

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 the application to avoid this problem. Another aspect of the mismatch between objects and processes is the fact that CORBA applications are often not pure object-oriented programs, i.e., an object's state might depend on information (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: 1) application-level state, consisting of the values of data structures within the CORBA object, 2) 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 3) 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.

Because a CORBA object's state is not contained entirely within the object, other parts of the object's process 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—need 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 interfaces and not ORB implementations, which means that ORBs can differ widely in their internals. Furthermore, ORB vendors tend to regard (and 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 [23]. 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.

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 some ORB/POA-level state. The vendor-dependent, nonstandard 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. Another consequence of the vendor-dependent 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.

3.3 Interaction with Unreplicated Objects

Traditional group communication systems often assume a homogeneous environment, where all of the application's components execute on top of the same fault-tolerant infrastructure. In practice, distributed CORBA 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 behind firewalls. Thus, there is a need to support two-way interoperable communication between fault-tolerant (replicated) applications and non-fault-tolerant (unreplicated) entities.

This interoperability requirement increases the complexity of the fault-tolerant infrastructure. 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.

Gateways [26] alleviate the complexity of interfacing replicated CORBA objects with unreplicated objects or with ORBs from other vendors. The part of the application/system that the fault-tolerant CORBA infrastructure supports and renders fault-tolerant, is referred to as a *fault-tolerant domain*. A gateway serves to bridge non-fault-tolerant clients/servers into a fault-tolerant domain by translating 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 [7]. Another use for gateways is to bridge two fault-tolerant CORBA infrastructures, where each uses different protocols or mechanisms.

Support for heterogeneity in FT-CORBA has a price, as end-to-end reliability cannot be guaranteed in all cases when unreplicated or non-fault-tolerant entities are involved. Despite these issues, gateways are useful in many scenarios where fault-tolerant CORBA applications must necessarily deal with clients outside of their reach. Wherever possible, we recommend deploying all of the clients and servers of a CORBA application over the same fault-tolerant infrastructure for reasons of efficiency and strong replica consistency.

3.4 Multitiered Applications

CORBA objects are often not restricted to either a pure client or a pure server role, but can act as clients for one

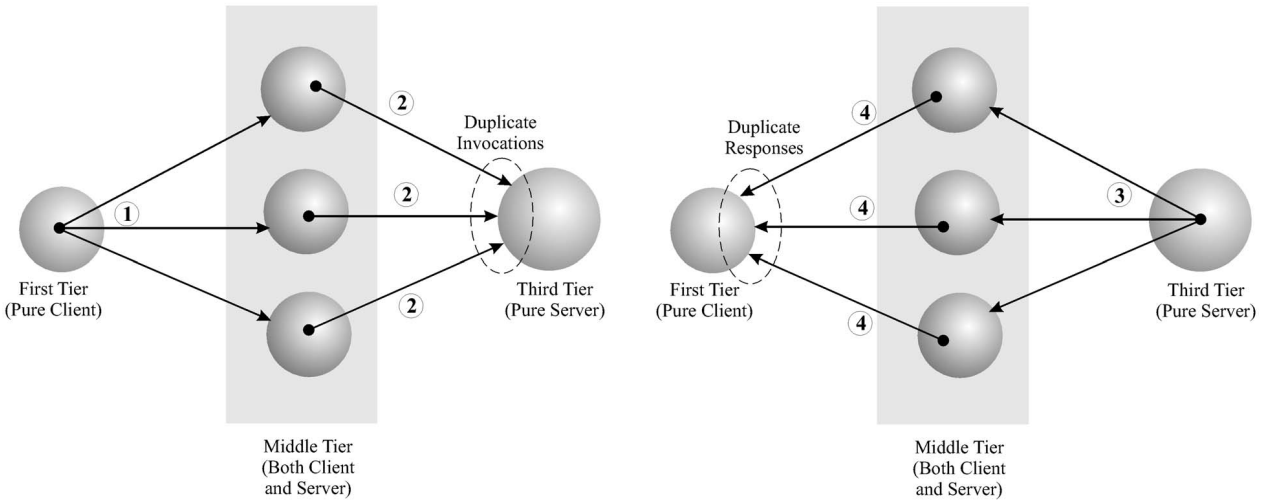

Fig. 4. Duplicate invocations and duplicate responses with an actively replicated middle-tier.

operation and servers 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 occurs in a multitiered application, where middle-tier objects (also known as application servers or business logic) mediate interactions between front-end pure clients and back-end pure servers. As a part of processing an incoming invocation, the middle-tier server might act as a client and, in turn, invoke the back-end server; the second invocation is known as a *nested operation*. A special case of this is a *callback*, where the middle-tier server, in its role as a client, invokes itself.

Nested invocations force us to consider replicating the clients (instead of the servers alone) of distributed applications. For fault tolerance, the middle-tier of a three-tier application needs to be replicated; however, this involves replicating *both* the client-side and the server-side portions of the middle-tier. While duplicate invocations are normally handled for server replication, multitiered applications require the handling of duplicate responses as well.

Consider the three-tier application in Fig. 4, where, for the sake of simplicity, only the middle-tier is actively replicated (the problem is no different even if all tiers are replicated). Each middle-tier replica processes the incoming invocation from the first-tier client (1) and, in turn, invokes the third-tier server (2). Each second-tier replica sends an identical invocation to the third-tier; unless two of these three duplicate invocations are suppressed, the state of the third-tier server might be corrupted by its processing of the same operation thrice (if this invocation represented the withdrawal of funds from a bank account, then the account would be debited three times, instead of only once). The third-tier server responds to the replicated middle-tier (3); in turn, each middle-tier replica sends an identical response to the first-tier client (4). Again, the duplicate responses from the second-tier 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.

Duplicate messages are equally possible for passively replicated objects under the case of recovery. Suppose that the middle-tier of Fig. 4 is passively replicated. If the primary middle-tier replica fails after invoking (and before receiving the corresponding response from) the third-tier server, a new middle-tier primary replica will be elected. Because the new primary cannot always ascertain the status of the invocation issued to the third-tier, the new primary will reissue the same invocation. The third-tier object, as the recipient of two identical invocations (from the old and new primaries, respectively), risks the corruption of its state if it processes both invocations. Thus, the third-tier and the first-tier objects need to detect and suppress duplicate messages to handle a passive replicated middle-tier.

In an asynchronous distributed system, it is difficult to guarantee the detection of duplicates at their source; duplicate suppression should also occur at the destination to handle duplicates that escape detection at their respective sources.

FT-CORBA implements duplicate detection by embedding unique, request-specific information in the "service-context" field of each IIOP request message. For this to work, the client-side ORB hosting the request's originator (the first-tier client) must create this context and encapsulate it into requests sent to the middle-tier; in the case of a nested invocation, the client-side ORBs of the middle-tier must forward this context. To exploit this duplicate-detection mechanism, objects in a nested-invocation chain must run on top of an FT-CORBA-aware ORB. Fault-tolerant CORBA systems that do not have access to an FT-CORBA-aware ORB must resort to some other mechanism (e.g., library interpositioning, client-side smart proxies, portable interceptors) to support duplicate detection and suppression for unreplicated clients. Of course, the performance overhead of accomplishing the service-context addition outside of the ORB is typically greater than when an FT-CORBA-aware ORB is used.

3.5 Nondeterminism

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 nondeterminism include the use of local timers, operating system-specific calls, processor-specific functions, shared memory primitives, etc.

Nondeterministic 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 nondeterministic applications. There is some truth in this 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 nondeterministic behavior of the replicated object are completely contained and do not wreak havoc on the replica consistency of the object.

However, there do exist situations where passive replication does not compensate for nondeterminism. This is particularly true of scenarios where the nondeterministic 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 nondeterministic 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 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 servers, both of whose states might be updated.

For passive replication to resolve nondeterministic behavior, there should be no persistent effect (i.e., no lingering "leakage" of nondeterminism) 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 nondeterministic decisions/inputs or if all accesses are performed in the context of a transaction aborted upon failure [11]. In general, though, passive replication is no cure for nondeterminism.

3.6 Identity and Addressing

CORBA is known for its weak identity model.[3] However, 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 an object's replicas. CORBA's location transparency also poses a problem because fault-tolerant systems often take advantage of their knowledge of the replicas' physical placement, e.g., to elect a new primary upon the failure of an existing primary replica, to optimize distributed protocols, or to manage the group membership of collocated components efficiently.

Reliable infrastructures often rely on an indirect addressing model for replicated objects in order to hide the fact that the number, and locations, of the replicas can change over time. This requires the FT-CORBA infrastructure to maintain a custom group-oriented addressing scheme that maps a replicated object's reference, at runtime, onto the references of the individual replicas currently in the group. This group reference is made available to clients and encapsulates both the server group's identity as well as the individual addresses of its constituent replicas. However, an object's group reference enumerates only the replicas that exist at the time of group-reference generation; thus, a group reference can become obsolete, as replicas fail and are reintroduced. 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 references that clients hold.

When a client invokes a method using a replicated server's group reference, the FT-CORBA infrastructure translates this to invoke the same method on the individual server replicas whose addresses are present in the group reference. Clearly, if the group reference that the client holds is stale (i.e., the membership of the group has changed so that none of the replica references contained in the group reference represents an 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 reference.

The obsolescence of group references is typically solved by embedding, within a group reference, 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. The FT-CORBA infrastructure's Replication Manager (shown in Fig. 2) might itself act as this forwarding persistent entity. If a client tries to invoke a replica using an outdated group reference, the replica's FT-CORBA infrastructure can transparently update the reference held by the client through standard CORBA redirection mechanisms (such as a LOCATION_FORWARD message, where a recipient ORB can redirect an incoming invocation to another address, much in the way that a post office provides mail forwarding services).

3.7 Factories

One of the limitations of early fault-tolerant CORBA implementations was that they could not adequately support objects with methods that return object references. A classic example of this problem is illustrated by "factories" whose sole purpose is to create and destroy CORBA objects in response to client requests. After requesting the creation of a new CORBA object and then obtaining a reference to the

3. Every CORBA object can be associated with a 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.

newly created object from the factory, a client can subsequently invoke operations on that object.

Factories are a very common and useful paradigm in distributed programming, e.g., the Factory design pattern [13] that many CORBA systems use extensively. In fact, FT-CORBA makes explicit provision for a Generic Factory interface specifically for the purpose of instantiating and deleting replicas, as described in Section 2.4. Typically, when asked to create an object, the FT-CORBA 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. Of course, to accomplish this, the client must first have a reference to the factory.

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

One way of achieving this is to have the factory deal explicitly with group management. However, factories are typically written by the application programmer; adding group management to the 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 violates replication transparency; also, the replicas of the 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 among themselves to achieve the end-result of creating a replicated object and generating a group reference. If the 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 in normal operations, but are active in the coordinated creation of a replicated object.

Another way of implementing a replicated factory is to decouple the replicas of the factory and to perform the coordination of the factory replicas using a higher-level entity, such as the FT-CORBA Replication Manager (shown in Fig. 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 reference using the individual application replica references that it has received from the various factories and returns this group reference to the client that requested the creation of the replicated object.

Regardless of whether the factories or the Replication Manager generate the group reference, FT-CORBA provides some interfaces and mechanisms to deal with replica creation. Using the FT-CORBA Property Manager interface described in Section 2.4, the application programmer can register custom factories for each object in the application. While FT-CORBA's Generic Factory interface makes the creation of replicated objects relatively straightforward, it requires significant modifications to, and the redesign of, existing CORBA applications.

3.8 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 are 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 can 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 versus passive replication) in a resource-aware manner [24] 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 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 operational replica 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 CPU cycles on its respective processor. Thus, passive replication consumes cycles on fewer processors during normal operation, i.e., normal fault-free operations are not performed by every replica on its respective processor. For operations that are compute-bound, i.e., requiring 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 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 versus 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. In a wireless setting, other resources, such as battery-power, may also be considered in the choice of replication style.

Hybrid active-passive replication schemes [15] 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 combining both replication styles has been proposed in [8]. 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 replication styles can be used at the same time in a distributed application; a unique feature is that the replication style can be dynamically specified on a per-operation basis.

3.9 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 has 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 seemingly stateless (in terms of 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 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.

4 Conclusion

As applications become more distributed and complex, the likelihood of faults undoubtedly increases because individual processors and communication links can fail independently. For most of the decade since its inception, CORBA had no standard support for fault tolerance. Various research efforts were expended to remedy this deficiency, with each of the resulting systems using replication to protect applications from faults. The Fault-Tolerant CORBA (FT-CORBA) standard represents the marriage of several of these efforts and their insights and comprises the specifications necessary to replicate CORBA objects.

We strongly believe that *reliability should not be an afterthought*. 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 nondeterminism. 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 nondeterminism 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 states concurrently.

Investing the thought and the effort to plan ahead for reliability while designing a new application can eliminate the redesign and reimplementation of the application when fault tolerance does become an issue. Planning ahead might involve examining

1. the important system/application state, i.e., the data that will need to be protected despite faults in the system,
2. the appropriate granularity of the application's objects (because replicating many small objects might impact performance or resources),
3. the critical elements of processing, i.e., the processing or operations that will need to continue uninterrupted, despite faults in the system, and
4. the data flows within the system (because objects that communicate frequently might need to be colocated within the same process).

Unfortunately, the FT-CORBA standard, 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 rectify any problems that already exist in applications and, therefore, does not provide solutions for cases where applications may crash due to design faults; for example, with FT-CORBA, if one replica crashes due to a programming error, such as a divide-by-zero exception, all of the replicas will crash identically. Furthermore, if the network 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 reestablished. Other open issues, such as supporting the CORBA Component Model (CCM), combining fault tolerance and real-time, combining fault tolerance and security, combining replication and transactions, still remain to be addressed by the developers of fault-tolerant CORBA systems.

In this paper, we have shared our experiences and insights as users of multiple CORBA implementations, developers of fault-tolerant CORBA systems, and 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. While some of the insights 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. It is our sincere hope that FT-CORBA users/implementors, application developers, and ORB vendors 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 applications.

Acknowledgments

The authors would like to acknowledge the anonymous reviewers whose detailed comments and feedback greatly improved the layout, the content, and the quality of this paper. This work has been partially funded by US National Science Foundation CAREER grant CCR-0238381.

References

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

[2] L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," *Proc. Fault-Tolerant Computing Symp.,* pp. 3-9, July 1978.

[3] D.H. Craft, "A Study of Pickling," *J. Object-Oriented Programming,* vol. 5, no. 8, pp. 54-66, 1993.

[4] 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," *Proc. IEEE 17th Symp. Reliable Distributed Systems,* pp. 245-253, Oct. 1998.

[5] J.C. Fabre and T. Perennou, "A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach," *IEEE Trans. Computers,* vol. 47, no. 1, pp. 78-95, Jan. 1998.

[6] P. Felber, "The CORBA Object Group Service: A Service Approach to Object Groups in CORBA," PhD thesis, Swiss Federal Inst. of Technology, Lausanne, 1998.

[7] P. Felber, "Lightweight Fault Tolerance in CORBA," *Proc. Int'l Symp. Distributed Objects and Applications (DOA '01),* pp. 239-247, Sept. 2001.

[8] P. Felber, X. Défago, P. Eugster, and A. Schiper, "Replicating CORBA Objects: A Marriage between Active and Passive Replication," *Proc. Second IFIP WG 6.1 Int'l Working Conf. Distributed Applications and Interoperable Systems (DAIS '99),* pp. 375-387, June 1999.

[9] P. Felber, R. Guerraoui, and A. Schiper, "The Implementation of a CORBA Object Group Service," *Theory and Practice of Object Systems,* vol. 4, no. 2, pp. 93-105, 1998.
[10] P. Felber, B. Jai, R. Rastogi, and M. Smith, "Using Semantic Knowledge of Distributed Objects to Increase Reliability and Availability," *Proc. Sixth Int'l Workshop Object-Oriented Real-Time Dependable Systems (WORDS '01),* pp. 153-160, Jan. 2001.
[11] P. Felber and P. Narasimhan, "Reconciling Replication and Transactions for the End-to-End Reliability of CORBA Applications," *Proc. Int'l Symp. Distributed Objects and Applications (DOA '02),* pp. 737-754, Oct. 2002.
[12] R. Friedman and E. Hadad, "FTS: A High Performance CORBA Fault Tolerance Service," *Proc. IEEE Workshop Object-Oriented Real-time Dependable Systems,* Jan. 2002.
[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
[14] R. Guerraoui, P. Felber, B. Garbinato, and K. Mazouni, "System Support for Object Groups," *Proc. ACM Conf. Object Oriented Programming Systems, Languages, and Applications (OOPSLA '98),* Oct. 1998.
[15] H. Higaki and T. Soneoka, "Fault-Tolerant Object by Group-to-Group Communications in Distributed Systems," *Proc. Second Int'l Workshop Responsive Computer Systems,* pp. 62-71, Oct. 1992.
[16] IONA and Isis, *An Introduction to Orbix+Isis,* IONA Technologies Ltd. and Isis Distributed Systems, Inc., 1994.
[17] S. Maffeis, "Run-Time Support for Object-Oriented Distributed Programming," PhD thesis, Univ. of Zurich, Feb. 1995.
[18] C. Marchetti, M. Mecella, A. Virgillito, and R. Baldoni, "An Interoperable Replication Logic for CORBA Systems," *Proc. Second Int'l Symp. Distributed Objects and Applications (DOA '00),* pp. 7-16, Feb. 2000.
[19] G. Morgan, S. Shrivastava, P. Ezhilchelvan, and M. Little, "Design and Implementation of a CORBA Fault-Tolerant Object Group Service," *Proc. Second IFIP WG 6.1 Int'l Working Conf. Distributed Applications and Interoperable Systems,* June 1999.
[20] L. Moser, P. Melliar-Smith, and P. Narasimhan, "Consistent Object Replication in the Eternal System," *Theory and Practice of Object Systems,* vol. 4, no. 2, pp. 81-92, 1998.
[21] 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," *Comm. ACM,* vol. 39, no. 4, pp. 54-63, Apr. 1996.
[22] *Distributed Systems,* S. Mullender, ed., second ed., chapters 7 and 8. Addison-Wesley, 1993.
[23] P. Narasimhan, "Transparent Fault Tolerance for CORBA," PhD thesis, Dept. of Electrical and Computer Eng., Univ. of California, Santa Barbara, Dec. 1999.
[24] P. Narasimhan and C.F. Reverte, "Configuring Replication Properties through the MEAD Fault-Tolerance Advisor," *Proc. Workshop Object-Oriented Real-Time Dependable Systems,* Oct. 2003.
[25] P. Narasimhan, K.P. Kihlstrom, L.E. Moser, and P.M. Melliar-Smith, "Providing Support for Survivable CORBA Applications with the Immune System," *Proc. 19th IEEE Int'l Conf. Distributed Computing Systems,* pp. 507-516, May 1999.
[26] P. Narasimhan, L. Moser, and P.M. Melliar-Smith, "Gateways for Accessing Fault Tolerance Domains," *Proc. Middleware 2000,* Apr. 2000.
[27] B. Natarajan, A. Gokhale, S. Yajnik, and D.C. Schmidt, "Doors: Towards High-Performance Fault Tolerant CORBA," *Proc. Second Int'l Symp. Distributed Objects and Applications (DOA '00),* pp. 39-48, Feb. 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, Dec. 2001.
[30] Object Management Group, "Portable Interceptors (Final Adopted Specification)," OMG Technical Committee Document formal/01-12-25, Dec. 2001.
[31] Object Management Group, "The Common Object Request Broker: Architecture and Specification, 2.6 Edition," OMG Technical Committee Document formal/02-01-02, Jan. 2002.
[32] G. Parrington, S. Shrivastava, S. Wheater, and M. Little, "The Design and Implementation of Arjuna," *USENIX Computing Systems J.,* vol. 8, no. 3, pp. 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, Univ. 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,* vol. 28, no. 9, pp. 963-979, July 1998.
[36] R. van Renesse, K.P. Birman, and S. Maffeis, "Horus: A Flexible Group Communication System," *Comm. ACM,* vol. 39, no. 4, pp. 76-83, Apr. 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,* vol. 4, no. 2, pp. 73-80, 1998.
[38] W. Vogels, R.V. Renesse, and K. Birman, "Six Misconceptions about Reliable Distributed Computing," *Proc. Eighth ACM SIGOPS European Workshop,* Sept. 1998.
[39] Y.M. Wang, Y. Huang, K.P. Vo, P.Y. Chung, and C.M.R. Kintala, "Checkpointing and Its Applications," *Proc. 25th IEEE Int'l Symp. Fault-Tolerant Computing,* pp. 22-31, June 1995.

Pascal Felber received the PhD degree in computer science from the Swiss Federal Institute of Technology and spent a few years at Oracle Corporation and Bell-Labs in the USA. He is an assistant professor at EURECOM, a leading international graduate school and research center in communications systems. His main research interests are in the area of object-based and dependable distributed systems. He is a member of the ACM and the IEEE.

Priya Narasimhan received the PhD degree from the University of California, Santa Barbara, and spent a few years commercializing the results of her research by serving as the CTO and vice-president of a start-up company. She is an assistant professor in the Electrical and Computer Engineering Department at Carnegie Mellon University. Her research interests include dependable distributed middleware systems, particularly where multiple properties, such as fault tolerance, real-time, and security, are required. She is a member of the ACM and the IEEE.