SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **33**:767–797 (DOI: 10.1002/spe.525)

Three-tier replication for FT-CORBA infrastructures

Roberto Baldoni and Carlo Marchetti*,†

Dipartimento di Informatica e Sistemistica, Università di Roma 'La Sapienza', Via Salaria 113, 00198 Roma, Italy

SUMMARY

Enforcing strong replica consistency among a set of replicas of a service deployed across an *asynchronous* distributed system in the presence of crash failures is a real practical challenge. If each replica runs the consistency protocol bundled with the actual service implementation, this target cannot be achieved, as replicas need to be located over a *partially synchronous distributed system* to solve the distributed agreement problems underlying strong replica consistency.

A three-tier architecture for software replication enables the separation of the replication logic, i.e. protocols and mechanisms necessary for managing software replication, from both clients and server replicas. The replication logic is embedded in a middle-tier that confines the need of partial synchrony and thus frees replica deployment.

In this paper we first introduce the basic concepts underlying three-tier replication. Then we present the interoperable replication logic (IRL) architecture, a fault-tolerant CORBA compliant infrastructure. IRL exploits a three-tier approach to replicate stateful deterministic CORBA objects and allows object replicas to run on object request brokers from different vendors. A description of an IRL prototype developed in our department is proposed along with an extensive performance analysis. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: software replication; architectures for dependable services; fault-tolerant CORBA

1. INTRODUCTION

1.1. Context

Software replication is a well-known technique used to increase the availability of a service exploiting specialized software running on commercial-off-the-shelf (COTS), cheap hardware. The basic idea underlying software replication is to replicate a server on different hosts connected by a communication network, so that the service's clients can connect to different *replicas* to obtain replies. Software replication is commonly implemented using two-tier (2T) architectures in which clients

*Correspondence to: Carlo Marchetti, Dipartimento di Informatica e Sistemistica, Università di Roma 'La Sapienza', Via Salaria 113, 00198 Roma, Italy.
†E-mail: marchet@dis.uniroma1.it

Received 1 March 2002
Revised 27 June 2002
Accepted 4 January 2003
Published online 15 May 2003
Copyright © 2003 John Wiley & Sons, Ltd.

directly interact with replicas. If the service is *stateless*, i.e. a result to a request only depends on the request message content, no replica consistency must be guaranteed and improving service availability mainly reduces to addressing the issue of letting clients invoke multiple replicas.

When dealing with the replication of a *stateful* service, system designers face the problem of guaranteeing a given degree of *consistency* among the local states of the replicas despite concurrent client requests and failures. *Active* [1], *passive* [2] and *semi-passive* [3] replication are well-known approaches to increase the availability of a stateful service while maintaining *strong* replica consistency. In these approaches, each replica embeds the service implementation and uses group communication primitives and services such as *total order multicast*, *view synchronous multicast*, *group membership*, etc., to ensure strong replica consistency and to keep a consistent view of the members of a group. Therefore, in these architectures, getting strong replica consistency passes through the implementability of such primitives in a given distributed system setting.

1.2. Motivation

Unfortunately, it is well-known that the implementation of these primitives relies on distributed agreement protocols [4,5]. As a consequence, implementing software replication through a 2T architecture requires replicas, and occasionally clients, to be deployed within a *partially synchronous* distributed system to overcome the Fisher, Lynch and Patterson (FLP) impossibility result [6]‡. A partially synchronous distributed system is an asynchronous distributed system with some additional timing assumptions on message transfer delays, process execution steps, etc. [9]. Partial synchrony can be easily guaranteed on small distributed systems deployed over either a LAN or a CAN (controlled area network).

Therefore, enforcing strong replica consistency among a set of replicas of a service deployed across an *asynchronous* distributed system, such as the Internet, in the presence of crash failures and/or partitioning is a relevant practical problem. This type of deployment would free the application designer from any constraint on where to place replicas, other than providing a valid building block to the solution of problems like survivability, service security, etc.

In the past, an answer to this problem has been given by either relaxing the consistency criterion enforced on replicas [10–12] or by using group toolkit primitives with probabilistic service guarantees [8]. As examples, group toolkits such as SPREAD [10] and TRANSIS [12] allow group partitioning and progress within each sub-group, but they require an *application-dependent* reconciliation phase after partition merging. The Xpand toolkit introduces ‘weak service levels’ which ensure that consistency criteria are weaker than the strong replica consistency [11].

1.3. Aim of the paper

In this paper we introduce a three-tier (3T) architecture for software replication. This architecture gives an answer to the problem of replica deployment in asynchronous distributed systems by separating the

‡Let us also remark that, under a practical point of view, the implementation of group primitives in partially synchronous distributed can suffer from unexpected slow down of processes and reduction of network bandwidth which can cause instability to the whole system [7,8].

Copyright © 2003 John Wiley & Sons, Ltd.

protocols and mechanisms necessary to manage software replication (i.e. the replication logic) from both the clients and server replicas. The replication logic is deployed within a middle-tier deployed *between* clients and server replicas. The rationale behind this approach is to confine the need of partial synchrony to the middle-tier in order to free the replicas' deployment and keep it strongly consistent. As a consequence, server replicas can be deployed on very distant sites interconnected by asynchronous point-to-point channels to the middle-tier. We also present a fault-tolerant CORBA (FT-CORBA) compliant system, namely the Interoperable Replication Logic (IRL) that exploits a 3T architecture to increase the availability of CORBA objects.

More specifically, the remainder of the paper is structured as follows. Section 2 points out the limitations of 2T architectures, introduces a desirable architectural property for software replication and presents the 3T architecture, discussing advantages and limitations. Section 3 introduces the IRL architectural design. IRL is a software infrastructure compliant to the FT-CORBA specification [13] that adopts a 3T approach for providing transparent replication of stateful deterministic CORBA objects. IRL exploits CORBA's interoperability and 3T replication to overcome the FT-CORBA '*common infrastructure*' limitation. Furthermore, by providing clients running on standard object request brokers (ORBs) with transparent access to replicated objects, IRL also overcomes the FT-CORBA '*legacy ORBs*' limitation. For the sake of completeness, Section 3 also includes an overview of the FT-CORBA standard's extensions and modifications to CORBA. Section 4 describes an IRL prototype, developed in our department, which represents a first proof of concept of a 3T architecture for software replication applied to build interoperable FT-CORBA infrastructures. Section 5 shows a comprehensive performance analysis of the prototype and describes some lessons learned. Section 6 describes the related work and compares IRL with other systems providing fault tolerance in CORBA. Finally, Section 7 concludes the paper.

2. SOFTWARE REPLICATION

In this section we first deal with the system model and with the main problem underlying software replication, i.e. strong replica consistency. Secondly, we describe two of the main software replication techniques, i.e. active and passive replication. Then we introduce the *client/server asynchrony* desirable property for software replication and discuss its impact on current techniques, architectures and tools supporting the implementation of strong replica consistency. Finally, we introduce the basic concepts on 3T software replication and discuss its advantages and limitations.

2.1. System model

We consider a service implemented by a set of deterministic replica processes (replicas) accessed by a set of client processes (clients). Processes can fail by crashing. After a crash event a process stops executing any action. A process is *correct* if it never crashes. Processes communicate through asynchronous eventually reliable channels. Messages are not duplicated and are delivered at most once. Messages sent by correct processes to correct processes are eventually delivered. We distinguish between the following system models.

- *Asynchronous distributed system model.* There is no known bound on the time a process takes to complete a task and on the delay introduced by communication channels to transfer messages.

Copyright © 2003 John Wiley & Sons, Ltd.

- *Partially synchronous distributed system model.* There is a bound (known or unknown) on the time a process takes to complete a task. There is a bound (known or unknown) on message transfer delays. These bounds can hold always, eventually, or only during *stability periods*. As examples, the timed asynchronous system model [14] and the asynchronous system augmented with unreliable failure detectors [4] are partially synchronous distributed system models.

2.2. Strong replica consistency

Strong replica consistency can be achieved by *linearizing* [15] the executions of a stateful replicated service. Informally speaking, linearizability ensures that each client operation takes effect on the replica state at some point between the operation invocation and its response, giving programmers the illusion that they are interacting with non-replicated objects. Under a practical point of view, a linearization can be obtained if replicas execute a set of updates in: (i) an ordered way; and (ii) an atomic way. Atomicity is satisfied if each state update is executed either by all or none of the server replicas; ordering is satisfied if each server replica executes state updates in the same order.

2.3. 2T replication techniques

A large number of replication techniques enforcing strong replica consistency have been proposed in the literature, e.g. [1–3,16]. These techniques are based on 2T architectures in which clients *directly* interact with server replicas embedding both group communication toolkits and the actual service implementation. As examples, we summarize here the active and passive replication techniques as they will be used in the next sections.

In active replication [1], a client sends a request to a set of *deterministic* server replicas. Each replica independently executes the request and sends back a reply to the client. To get linearizability, clients and servers must interact through a *total order* (or *atomic*) multicast primitive [17]. This primitive ensures that all server replicas process requests in the same order before failing.

In passive replication [2], a particular *primary* replica serves all client requests. Upon receiving a request, the primary processes the request, produces a result and reaches a new state. Then the primary sends an update message to the backups before returning the result to the client. Note that replicas can be non-deterministic as: (i) only the primary state evolves upon processing the client request; and (ii) the primary is in charge of updating the state of the backups. One way to enforce linearizability is to let the primary use a *view-synchronous multicast* primitive to send the updates. The implementation of this primitive passes through a protocol executed by the replicas in order to *agree* on the set of messages exchanged in a *view* [18].

2.4. A desirable property for software replication

Clients and server replicas implementing a replication technique should enjoy the following architectural property.

Client/server asynchrony. Clients (respectively server replicas) run over a distributed system without any timing assumptions (i.e. an asynchronous distributed system).

Copyright © 2003 John Wiley & Sons, Ltd.

If satisfied, *client/server asynchrony* allows clients and server replicas of a stateful service to be deployed within a system like the Internet.

2T replication techniques cannot satisfy the client/server-asynchrony property. These techniques indeed rely on the use of specific communication primitives among replicas, whose implementation boils down to solving an agreement problem [5]. Therefore, these techniques can be adopted *only if* the distributed system underlying replicas is *partially* synchronous, otherwise we face the FLP impossibility result [6].

Furthermore, as discussed in [7], group communication toolkits can only be used effectively in practice if the distributed system belongs to a well-managed environment where it is possible to control the loads of the network and CPUs, i.e. to control the major sources of asynchrony. Otherwise, a single slow member or an unexpected drastic reduction of network bandwidth can slow down the overall system leading to low service availability despite replication [7].

For these reasons, existing software replication techniques providing strong consistency guarantees are usually implemented in *clusters* of co-located and tightly coupled workstations interconnected by a LAN [19,20]. This reduces the resilience of these systems to site failures. For example, if the whole cluster site crashes or fires, both service availability and data integrity are definitively lost.

2.5. 3T software replication

The basic idea underlying three-tier software replications is to free asynchronous clients and replicas from enforcing strong replica consistency. This is achieved by embedding the replication logic within a software middle-tier *physically detached* from both clients and replicas. The middle-tier is responsible for ensuring the linearizability on the executions of server replicas forming the end-tier (see Figure 1(a)).

In this architecture, a client sends the request to the middle-tier, which forwards it to server replicas according to the replication logic implemented by the middle-tier. A simple message pattern is shown in Figure 1(b) in the case of active replication of the end-tier. The figure shows that each *deterministic* replica executes each client request according to a total order *defined by the middle-tier*. Results are then sent to the middle-tier, which finally returns them to the clients.

To ensure the termination of a client/server interaction in the presence of failures, the middle-tier has to be fault tolerant. In particular, if a middle-tier entity that was carrying out a client/server interaction crashes, another middle-tier entity has to conclude the job in order to enforce end-tier consistency despite failures. This implies that the middle-tier entities can maintain a strongly consistent replicated state. Depending on the replication technique implemented (e.g. active, passive or semipassive), the middle-tier state could include ordering information on client requests, the middle-tier entity responsible for handling each client request as well as its processing state, etc. Therefore, there is a need to run a distributed agreement protocol among the middle-tier entities. This implies assuming a partially synchronous model. However, partial synchrony is needed *only* among middle-tier entities. As a consequence the *client/server asynchrony* property is satisfied [25].

Let us also remark that 3T replication produces a sharp separation of concerns between the management of the replication and the actual service implementation. The middle-tier is specialized to handle the replication logic and confines the use of group communication primitives to a well-defined and controlled system region. This enables the reduction of asynchrony sources (see the previous

Copyright © 2003 John Wiley & Sons, Ltd.

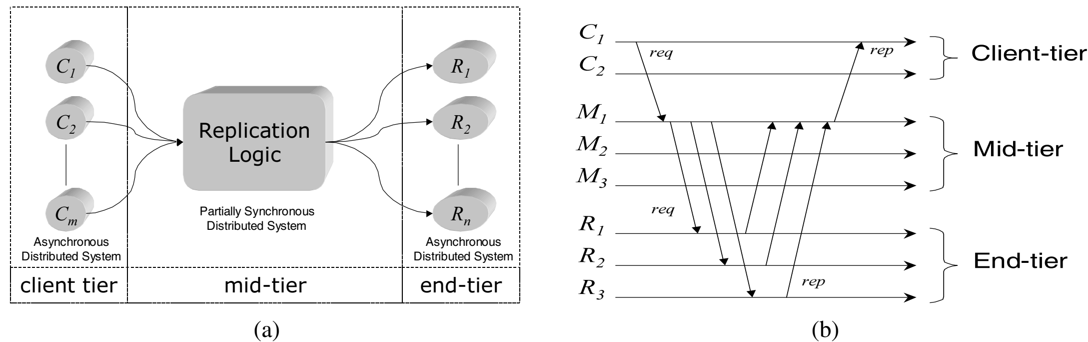


Figure 1. Three-tier architecture software replication: (a) a basic architecture; (b) a simple message pattern for end-tier active replication.

section) and group communications to be run in a nice setting obtaining the best performance in terms of stability and predictability [7,8].

3T replication also brings the following additional advantages.

- *Thin clients and servers*. Clients are only augmented with a lightweight request retransmission mechanism, a *necessary* tool to cope with middle-tier failures and arbitrary message transfer delays. This simplifies the implementation of client replication and failure transparency and widens the spectrum of devices that can host such clients, e.g. mobile phones, PDAs, etc. Analogously, replicas implement simple functionality, e.g. duplicate filtering, other than providing the actual service. Furthermore, clients and server replicas may communicate with middle-tier entities on a point-to-point basis following a simple request/response asynchronous message pattern: they do not need to implement any multicast primitive.
- *Loose client and replica coupling*. Clients and replicas are *loosely coupled*: they do not exchange messages among themselves. As consequences, 3T replication makes it easy to modify the replication logic without impacting the client code. It is also possible to implement on-the-fly changing of the replication style of the end-tier, e.g. from active to passive replication [21].
- *Decoupling service availability from data-integrity*. Separating the replication logic from the actual service implementation enables the preservation of data integrity despite site failures. Service availability is enforced as long as the middle-tier is live. Furthermore, developers are free to tune the system guarantees in terms of data-integrity and high-availability as they can select the number of middle-tier entities (determining the service availability level) and end-tier replicas (determining the data-integrity level) independently. Finally, it would also be possible to assume different failure models for the middle-tier and end-tier, e.g. a crash failure model for middle-tier elements and a byzantine failure model for end-tier replicas.
- *Middle-tier extensibility*. The middle-tier behaves as a centralized, highly fault-tolerant component. This simplifies the introduction of other desirable functionalities such as load balancing of client requests and exploiting semantic knowledge [22], with minimal or no impact on the replica code.

Copyright © 2003 John Wiley & Sons, Ltd.

Therefore, 3T replication allows nice deployments to be set up, in which a large number of thin clients access a service implemented by a large number of replicas deployed on a wide-area network through a small number of middle-tier entities running in a well-managed setting. The price to pay for these advantages are: (i) an additional hop in the client/server interaction; and (ii) the dependence of the service availability on the availability of the middle-tier.

Let us finally remark that a replication technique that satisfies the previous properties and characteristics allows clients and server replicas to be implemented over standard technologies based on TCP such as Internet inter-ORB protocol (IIOP) and SOAP.

3. INTEROPERABLE REPLICATION LOGIC

IRL exploits 3T replication to build an infrastructure providing transparent replication of stateful deterministic CORBA objects [23]. IRL provides developers of FT-CORBA applications with fault monitoring and replication management functionality, which are made accessible through interfaces that comply with the FT-CORBA specification [13].

The decision to implement a prototype of a 3T replication scheme compliant with the FT-CORBA specification is based on the following reasons: (i) FT-CORBA is the best-known standard on fault tolerance in distributed object technology; (ii) FT-CORBA suffers from some limitations that make it still appealing from a research perspective; and (iii) FT-CORBA extends CORBA clients with a request retransmission and redirection mechanism that fits the thin, replication-style-independent client model of a 3T replication system.

In the remainder of this section, we first summarize CORBA, the main FT-CORBA extensions to the CORBA specification and its limitations. Then we introduce the IRL FT-CORBA compliant design (see also [24] for additional details). Let us point out that even though the paper presents a FT-CORBA compliant platform as a case study, the notion of 3T replication is not related to a specific technology [25].

3.1. A brief overview of CORBA

CORBA [23] is a standard middleware platform for the development of distributed object-oriented applications. By decoupling object interfaces from their implementations, CORBA provides nice features such as interoperability and location transparency. The CORBA enabling technology is the Object Request Broker (ORB), which can be seen as a software bus interconnecting remote objects deployed over a network of heterogeneous computers. The ORB adopts standard interfaces to free developers from dealing with low-level issues such as operating systems and network calls. A basic client/server interaction in CORBA is as follows. Upon creation, a CORBA object publishes its interoperable object reference (IOR) which contains the information needed by clients to reach it (host IP address, port number, etc.). Once a client has retrieved an object's IOR, it can invoke operations on the object exploiting either the static or dynamic invocation interface mechanisms (SII and DII, respectively). The client request is then passed to the client ORB that is in charge of: (i) marshalling the request; and (ii) sending it to the destination ORB through IIOP. The destination ORB unmarshalls the request and delivers it to an object adapter, which finally passes the request to the actual object implementation. The latter can accept incoming requests either through the static skeleton interface

Copyright © 2003 John Wiley & Sons, Ltd.

or through the dynamic skeleton interface mechanisms (SSI and DSI, respectively). Once the object implementation produces a result, it is sent to the client following the inverse path.

This basic remote procedure call (RPC)-like invocation style can be extended and modified through further standard mechanisms according to the users' needs. As an example, it is possible to implement out-of-band signalling by putting information into *service contexts* that can be transparently piggybacked by the ORB onto request and reply messages. It is also possible to extend the basic request-handling functionality provided by the ORB by customizing portable interceptors [26,27], i.e. configurable hooks allowing the alteration of the standard ORB behavior during request processing. Customizing interceptors enables the implementation of request redirection and piggybacking in a straightforward and efficient way.

3.2. Overview of the FT-CORBA specification

FT-CORBA achieves fault tolerance of *deterministic* objects through object redundancy, fault detection and recovery. Replicas of a CORBA object are deployed on different hosts of a *fault tolerance domain* (FT-domain). A FT-domain is a collection of hosts interconnected by a non-partitionable computer network. Replicas of an object are called *members* as they are collected into an *object group*, which is a logical addressing facility allowing clients to transparently access the object group members as if they were a singleton, non-replicated, highly-available object [28]. If the replicated object is stateful, then developers can decide whether to explicitly handle replica consistency or to demand this task to the infrastructure. In the latter case *strong replica consistency* must be automatically enforced. The main FT-CORBA modifications and extensions to the CORBA standard concern *object group addressing*, a new *client failover semantic* and new mechanisms and architectural components devoted to *replication management*, *fault management* and *recovery management*.

3.2.1. Object group addressing

To identify and address object groups, FT-CORBA introduces *interoperable object group reference*s (IOGRs). An IOGR is a CORBA IOR composed of multiple *profiles*, each profile pointing to either: (i) an object group member, e.g. in the cases of passive and stateless replication; or (ii) a gateway orchestrating accesses to the object group members, e.g. in the case of active replication.

3.2.2. Extensions to the CORBA failover semantic

FT-CORBA compliant client ORBs provide failure and replication transparency to client applications by implementing the *transparent client reinvocation* and *redirection* mechanisms. These mechanisms consist of: (i) client ORBs uniquely identifying each client request by a unique REQUEST service context and using the destination IOGR to send the request to a group member (addressed by an IOGR profile); (ii) a client ORB receiving a *failover exception* (i.e. a CORBA COMM_FAILURE, TRANSIENT, OBJ_ADAPTER or NO_RESPONSE exception with completion status NO or MAYBE). Exploits another IOGR profile to re-issue the request until either a result is received or a timeout of duration equal to the *request expiration time* value (contained in the request identifier) has elapsed. Only in the latter case is a CORBA system exception thrown to the client application.

Copyright © 2003 John Wiley & Sons, Ltd.

3.2.3. Replication management

Replication management includes the creation and the management of object groups and object group members. The **ReplicationManager** (RM) component is responsible for carrying out these activities. In particular, when asked for object group creation, RM exploits *local factories* to create the members, collects the members' references and returns the object group reference. Local factories are provided by application developers and registered with the RM. This component allows the setting of a wide range of properties concerning the way an object group is replicated and monitored for failures. As an example, it is possible to select a replication technique, e.g. STATELESS or ACTIVE, for a group and choose if consistency among stateful group members has to be maintained by the application or by the infrastructure.

3.2.4. Fault management

Fault management concerns the detection of object group members' failures, the creation and notification of fault reports, and fault report analysis. These activities are carried out by the **FaultDetectors**, **FaultNotifier** (FN) and **FaultAnalyzer** components, respectively. **FaultDetectors** can be configured in a flexible manner in order to perform host-, process- and object-level failure detection. However, only object-level failure detection is specified: an object group member can be monitored for failures by a **FaultDetector** if it is *monitorable*, i.e. it implements the PullMonitorable interface that allows a **FaultDetector** to ping the object through the is_alive() method. This method is invoked periodically by object-level **FaultDetectors**, according to the timeout values specified by the FaultMonitoringIntervalAndTimeout property[§]. If an object does not return from the is_alive() method invocation within the specified timeout, its **FaultDetector** alerts the FN. The FN is based on the publish-and-subscribe paradigm: it receives object fault reports from the **FaultDetectors** (publishers) and propagates these fault notifications to the RM and other clients (subscribers).

3.2.5. Recovery management

Recovery management is based on two *mechanisms*, namely **logging** and **recovery**, that exploit two IDL interfaces (Checkpointable and Updateable). *Recoverable* application objects implement these interfaces to let **logging** and **recovery** mechanisms read and write their internal state. More specifically, the **logging** mechanism periodically stores in a log information such as member's state/state update, served requests, generated replies, request expiration times, etc., while in a log while the **recovery** mechanism retrieves this information from the log when, for example, setting the initial internal state of a new group member.

A FT-domain has to contain one logical instance of the RM and one of the FN. **FaultDetectors** are spread over the domain to detect failures of monitorable object group members and (optionally) processes and hosts. This software infrastructure is commonly referred to as a *fault tolerance infrastructure* (FT-infrastructure).

[§]This property can be set by application developers through the RM.

Copyright © 2003 John Wiley & Sons, Ltd.

FT-CORBA suffers from some limitations. In this work we focus on the so-called '*common infrastructure*' and '*legacy ORBs*' limitations [13].

- Common Infrastructure. FT-CORBA states that: (i) all the hosts of a FT-domain must use ORBs and FT-infrastructures from the same vendor; and (ii) the members of an object group must be hosted by ORBs and FT-infrastructures from the same vendor. The following sections show how this limitation is overcome by IRL through a 3T architecture.
- Legacy ORBs. FT-CORBA requires compliant client ORBs to extend their invocation semantic in order to deal with IOGRs, i.e. to implement the retransmission and redirection mechanism. As a consequence, non-compliant ORBs are not able to take full advantage of replication. This limitation has been overcome in IRL by designing a custom portable interceptor that implements the CORBA failover semantic without impacting on the application or ORB code. Further details concerning this issue can be found in [24,29].

Let us conclude this section by pointing out that FT-CORBA is the result of many research efforts carried out in the fields of distributed system fault tolerance and distributed object computing that will be discussed and compared to IRL in Section 6.

3.3. IRL architectural overview

IRL is the middle-tier of a 3T architecture for software replication of CORBA objects. This middle-tier includes both the middle-tier objects handling transparent replication of stateful CORBA objects and the highly available objects providing replication and fault-management functionality. Figure 2 illustrates the main IRL FT-infrastructure components which are implemented as standard CORBA objects running on top of unmodified ORBs.

IRL achieves *infrastructure portability and replica interoperability*, i.e. IRL is portable on different ORBs from different vendors and replicas can run on different ORBs from different vendors. In particular, IRL meets portability by exploiting only standard CORBA mechanisms such as SSI and DSI, portable interceptors, etc. This allows the migration of IRL code to different CORBA platforms with minimal effort. Furthermore, communication among clients, middle-tier components and object group members, i.e. end-tier server replicas, occur through standard CORBA invocations.

Clients of a stateful object group interact with the IRL object group handler (OGH) component which masters the interactions with object group members. In the case of stateless replication, clients connect directly to a single member. In both cases, communication occurs through the standard IIOP protocol. As a consequence, IRL supports clients and replicas running on heterogeneous ORBs overcoming the FT-CORBA common infrastructure limitation.

To get both the termination of client/server interactions and highly available management and monitoring services, IRL components, namely OGH, RM and FN, are replicated in Figure 2. These services maintain a replicated state, which is kept strongly consistent by exchanging messages through an *intra-component message bus*. This bus is based on a TCP/IP protocol stack and it can be implemented by a group communication toolkit in order to simplify the replication of the middle-tier. Therefore, only replicas of IRL components run within a partially synchronous system. Clients and object group members can run in a system without any underlying timing assumptions. As a consequence IRL satisfies the client/server asynchrony property.

Copyright © 2003 John Wiley & Sons, Ltd.

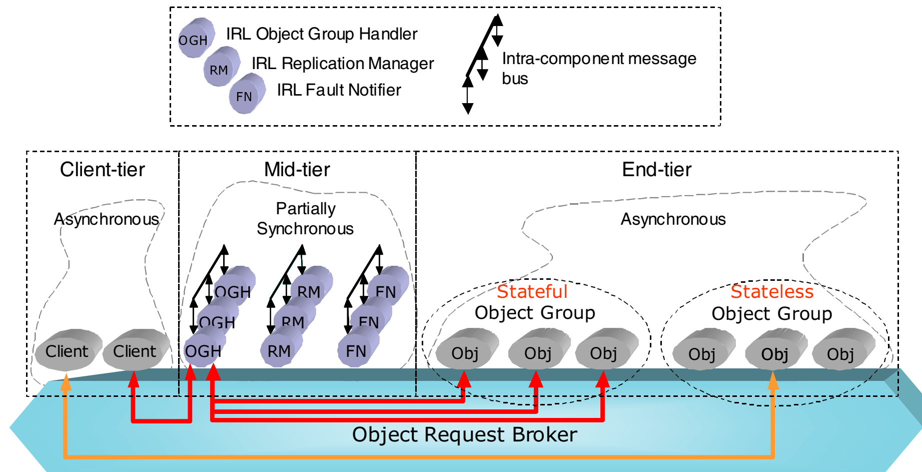


Figure 2. IRL architecture.

In the following, we present a short description of the main IRL components. The description is independent of the replication techniques adopted to increase the component's availability through replication, which will be discussed in Section 4.

3.3.1. IRL Object Group Handler (OGH)

An IRL OGH component is associated with each *stateful* object group: it stores the IORs of the group members and information about their availability. OGH is responsible for enforcing strong replica consistency among its group members' states. OGH is the actual middle-tier of a 3T replication architecture. By exploiting DSI, it adopts the interface of the object group members it is associated with. Then it accepts all incoming client connections, receives all the requests addressed to its object group, imposes a total order on them, forwards them to every object group member, gathers the replies and returns them to the client (see Section 4.3). Let us remark that: (i) neither clients nor replicas exploit group communication toolkits; and (ii) clients connect to distinct OGHs for accessing distinct stateful object groups. This favors the scalability of the approach with respect to the number of object groups.

3.3.2. IRL Replication Manager (RM)

This component is a FT-CORBA compliant RM. In particular, when RM is requested to create a new object group, it spawns new object group members invoking FT-CORBA compliant *local factories* and returns an object group reference. IRL RM allows the management of stateless and stateful object groups with application/infrastructure controlled membership and consistency. In the case of

Copyright © 2003 John Wiley & Sons, Ltd.

infrastructure controlled consistency, RM also spawns a replicated IRL OGH component and returns an object group reference pointing to OGH replicas.

3.3.3. IRL Fault Notifier (FN)

As a FT-CORBA compliant FN, the IRL FN receives: (i) subscriptions for failure notifications from its clients; (ii) object fault reports; and (iii) heartbeats for host failure detection. The last two types of messages are both sent from IRL local failure detectors (LFDs). A LFD is an object-level FT-CORBA compliant FaultDetector. A single LFD process runs on each host of an FT-domain[¶]. Upon receiving an object fault report from a LFD, the FN forwards it to the RM and to clients that had subscribed to the faulty object. In addition to this, the FN and LFDs implement host-level failure detection as each LFD periodically sends heartbeats to the FN according to a user-configurable property. Upon not receiving a heartbeat within a user-configurable timeout value, the FN creates a host fault report that is pushed to the RM as well as to every client that had subscribed to objects running on the faulty host.

To run IRL in a given FT-domain, it suffices to install the **IRL Factory** component on each FT-domain host which is able to launch on its host new IRL OGH, RM, FN and LFD component replicas.

4. AN IRL PROTOTYPE

The IRL design described in the previous section can be implemented in ways that mainly differ in:

1. the techniques and technologies adopted to achieve infrastructure fault tolerance;
2. the protocol run by the OGH replicas to enforce consistency of the object group members.

Therefore, this section points out the techniques adopted by the IRL prototype, developed in our department, to address the aforementioned points. More specifically, the choice of the most suitable replication technique for each IRL FT-infrastructure component is based on the state it maintains and on its deployment and fault-tolerance requirements. To simplify the explanation, we identify two classes of components, host-specific components (LFD and IRL Factory) and domain-specific components (OGH, RM and FN).

4.1. Host-specific components

The LFD and IRL Factory components are installed locally on each host and their activities are related only to that host. As a consequence, such components do not need to survive if their host crashes. However, being subject to software failures, they are locally replicated. In particular, IRL Factory is stateless and is replicated by running on every host two distinct IRL Factory processes that exchange *I'm alive* messages to check the liveness of the partner and recreate it upon detecting a failure.

[¶]An LFD is responsible of monitoring the monitorable objects running on its host according to the timeout values specified by application developers through RM.

Copyright © 2003 John Wiley & Sons, Ltd.

LFDs maintain a simple state composed by the references of the monitorable objects running on its host. For this reason, LFD logs its state on stable storage upon each state change and is monitored for failures by the IRL Factory running on its host, which launches a new LFD replica upon detecting a LFD software crash event. The new LFD instance re-establishes the availability of the monitoring service by using the log of the crashed one.

4.2. Domain-specific components

IRL RM, FN and OGH components implement functionality that must survive host crashes. In this section we describe a generic *wrapper-based* technique to replicate a *non-deterministic* CORBA object. This technique is shown in the case of RM replication. The OGH replication is explained in the next section.

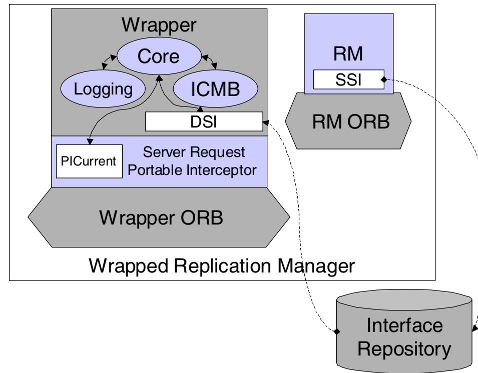
RM invokes local factories upon receiving requests for group creation, modification and disposal. This makes RM non-deterministic. Such non-determinism is due to the unpredictability of the results that local factories return to RM, e.g. IORs of new object group members. This leads us to adopt *passive replication* as the RM replication technique. To implement this technique, each RM replica is wrapped by a Wrapper CORBA object. Wrapper accepts all the requests to an object and implements an intra-component message bus in order to synchronize wrapped backup replicas' states with the primary one.

The Wrapper internal architecture is shown in Figure 3(a). Wrapper is a CORBA object that can use DSI to adopt the interface of the wrapped object, in this case RM[‖]. It also uses a server request portable interceptor to: (i) read the FT-CORBA REQUEST service context identifying the incoming requests; and (ii) perform request redirection when necessary, e.g. backups redirect client requests to the primary. Such mechanisms exploit the portable interceptors' PICurrent data structure [27], which allows CORBA applications to exchange information with the underlying interceptors[**]. Wrapper implements the following modules (see Figure 3(a)).

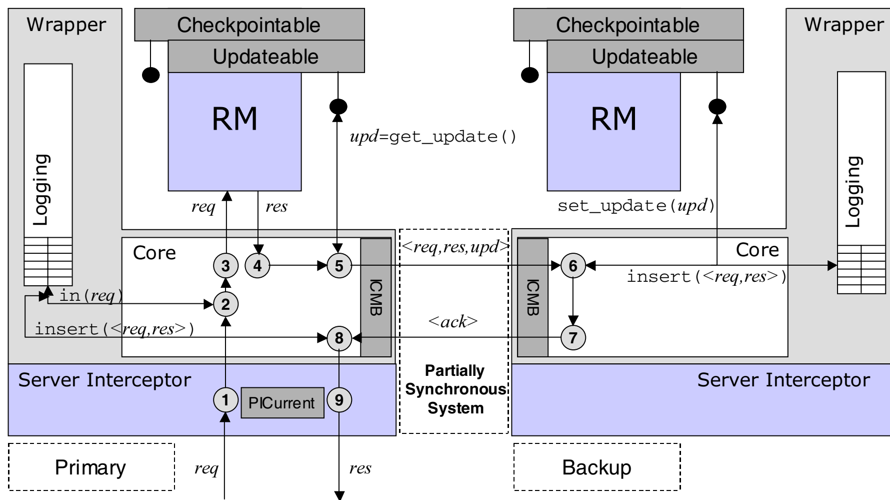
- logging. This module logs ⟨*request*, *reply*⟩ pairs and performs garbage collection on the basis of the request expiration time. logging is used by core to identify multiple identical invocations and, then, to return the same result to the same request.
- intra-component message bus (ICMB). This module provides core with the primitives and services necessary to manage passive replication, i.e. a group membership service and a view-synchronous multicast primitive (*vs-multicast*; see also Section 2.3). The current ICMB prototype embeds the Jgroup group toolkit [30].
- core. This module implements the passive replication protocol using functionality provided by the former modules and interacting with the underlying interceptor.

[‖]If the wrapped object interface is known at compile-time, Wrapper can also be configured to statically adopt a given interface through a SSI.
[**]Let us point out that RM runs over an ORB distinct from the wrapper ORB to avoid recursive interception of requests forwarded by the wrapper to its RM replica [26]. The RM ORB is co-located with the wrapper ORB. Practice demonstrates that these two co-located ORBs do not fail independently.

Copyright © 2003 John Wiley & Sons, Ltd.



(a)



(b)

Figure 3. Replication of the RM component: (a) wrapper-based architecture; (b) main steps of a client–RM interaction.

Copyright © 2003 John Wiley & Sons, Ltd.

The architecture works as follows. The core handles two types of events: *view changes*, triggered by ICMB upon detecting a change in the membership of RM replicas, and the *arrival of a request* from the DSI layer.

Upon view change, core first checks if it is the primary. In the affirmative, it invokes ICMB to *vs-multicast* an *IOR request* message to each backup member in order to build the new IOGR. As soon as all backups reply, the primary core builds a new IOGR that it *vs-multicast*s to backups. At the end of this phase, each backup core puts the new IOGR in PICurrent and sets a primary flag to false. This informs the interceptor to redirect incoming requests to the primary wrapper, whose profile is in the new IOGR. The primary wrapper core sets primary true. This causes the interceptor to let client requests reach the primary wrapper DSI layer. Hence the primary can start processing incoming requests (see Figure 3(b)).

Once a request *req* arrives at the primary (step 1), the following algorithm is executed.

- The interceptor first extracts the REQUEST service context from the request and puts it into PICurrent, then it relays the control to the DSI layer that parses the request and notifies core of the request arrival.
- core fetches the request identifier from PICurrent and then, by invoking logging, checks if *req* has already been executed by RM (step 2). In the affirmative, it immediately returns the logged result to the client, otherwise it forwards *req* to RM for processing (step 3).
- As soon as RM has performed the operation producing a result (*res*, step 4), core invokes the get_update() method of RM[††]. If the update (*upd*) is not empty (i.e. the operation modified the RM internal state) core invokes ICMB to *vs-multicast* an update message composed by a triple $\langle req, res, upd \rangle$ (step 5). Otherwise it just returns *res* to the client (step 9).
- Upon receiving an update from ICMB, each backup core updates its RM replica and its own state: it invokes the set_update(*upd*) method of RM and inserts the $\langle req, res \rangle$ pair into logging (step 6).
- Once the primary exits its *vs-multicast* invocation, it updates its own state by inserting the $\langle req, res \rangle$ pair into logging (step 8). Finally, the *res* is returned to the client (step 9).

Let us remark that the presented technique can be used to passively replicate **any** CORBA object using a quite common group communication system and without impacting on the object internal code.

4.3. IRL 3T replication protocol

In the current IRL prototype, OGH implements a simple 3T replication protocol that adopts active replication as its end-tier replication style, i.e. all object group members process all client requests in the same order before crashing. The middle-tier adopts passive replication, i.e. at most one OGH replica at a time accepts incoming client requests.

The overall protocol supports static groups of both OGH replicas and object group members and is based on the following constraints, mechanisms and properties on components and objects.

[††]RM implements both the Checkpointable interface—exporting get_state() and set_state() methods—and the Updateable interface—exporting get_update() and set_update() methods. The former is exploited by the wrapper to perform full state transfer, e.g. when a new member joins the group.

Copyright © 2003 John Wiley & Sons, Ltd.

4.3.1. Client-tier

Clients implement the simple FT-CORBA request redirection and retransmission mechanisms. Non-FT-CORBA compliant client ORBs are augmented with the IRL object request gateway (ORGW) component. ORGW is a CORBA client request portable interceptor that emulates FT-CORBA client-side mechanisms in a portable way, without impacting on the ORB or the client application code. This allows IRL to overcome the FT-CORBA legacy clients limitation. Additional details about ORGW can be found in [24,29].

4.3.2. End-tier

Each stateful object group member has to do the following.

- Be *deterministic*, to fail by crashing and it cannot invoke other objects in order to reply to a request.
- Be wrapped by the incoming request gateway (IRGW) IRL component to avoid multiple executions of the same request due to either middle-tier failures or the asynchrony of the underlying system (i.e. IRGW ensures at most once semantic). Similarly to the logging function presented in Section 4.2, IRGW implements a simple logging mechanism of $\langle request, result\rangle$ pairs. IRGW also implements a get_last method used by the OGH replication protocol (see below). This method returns the sequence number of the last request received by IRGW from OGH.
- Implement at least the Checkpointable and optionally the Updateable FT-CORBA interfaces to execute state transfers and synchronizations.

4.3.3. Fault Notifier

FN performs:

- *perfect failure detection*, i.e. accurate and complete in the sense of [4], with respect to OGH replicas: FN does not make mistakes when detecting middle-tier host crashes[‡‡];
- *eventual strong complete* failure detection with respect to object group member failures, i.e. FN can make mistakes by erroneously detecting some object group member failures as members can be deployed within an asynchronous system[*].

We make no assumption on the *accuracy* of FN with regards to object group member failures as they run within an asynchronous distributed system [4].

[‡‡]Perfect failure detection can likely be enforced by running OGH replicas either in a setting with the necessary amount of synchrony (e.g. a CAN) or in a partially synchronous system (e.g. a timed asynchronous system) with hardware watchdogs [31]. The former option is used by the current IRL prototype.

[*]Actually, FN can make mistakes *only* by erroneously detecting a host failure, as only host failures are detected using heartbeats. In contrast, object-level failure detection is performed on a local basis as each LFD monitors objects running on its host. As a consequence, object fault reports sent by LFD to FN can be assumed as reliable.

Copyright © 2003 John Wiley & Sons, Ltd.

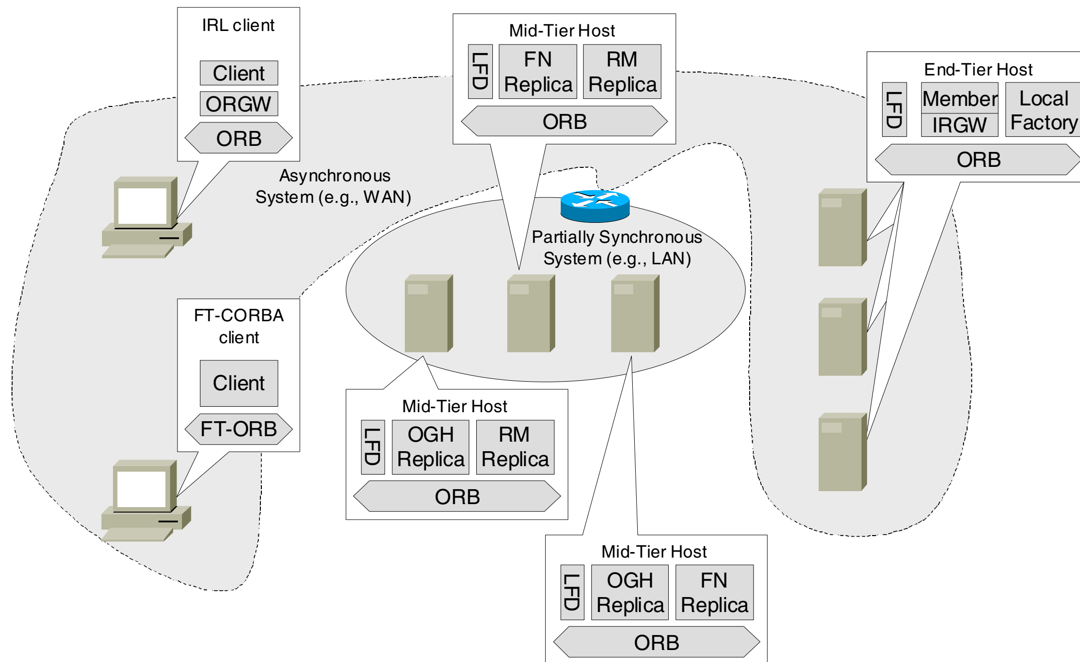


Figure 4. An example of IRL deployment.

Figure 4 illustrates an example of IRL deployment. For completeness, we also show RM replicas and local factories.

4.3.4. OGH protocol

When RM is requested by a client to create a stateful object group, it spawns a set of OGH replicas running on distinct hosts of the middle-tier and the object group members. Then, RM returns an IOGR pointing to OGH replicas to the client, so that the following client requests reach OGH replicas. At that time the *primary* OGH starts to accept incoming client requests. It forwards each request to all the object group members along with a local sequence number increasing for each new client request. Members process requests and send results back to the OGH primary. Once the primary has received the result from all the members, it forwards the result to the client. If an end-tier replica is suspected, FN notifies the primary which consequently stops waiting for the reply from the suspected replica. This suffices to enforce atomicity and ordering as long as the primary is correct. On the other hand, when a primary crashes, the last client request forwarded by the primary may not reach some member, provoking, thus, an atomicity violation. In this case, FN notifies backups of the primary OGH crash and backups select a new primary according to a deterministic criterion based on the list of currently

Copyright © 2003 John Wiley & Sons, Ltd.

active OGH replicas. The new primary is in charge of restoring consistency among the members' states, performing a *recovery protocol*. During this protocol, the new primary first invokes the get_last() method on the IRGW of each member. In this way the new primary obtains the sequence number of the last request served by each member. Then it selects the most updated member (MUM) which has the maximum sequence number (MSN), gets the state of the MUM—comprising the IRGW state—and sets this state into those members that have a sequence number lower than MSN[†]. At the end of this protocol all the members are consistent and the new primary starts serving client requests. Further details on the protocol implemented by OGH are described in [32].

Let us finally note that this protocol actually represents a first proof of concept of the three-tier replication scheme suitable for an experimental analysis and, as such, suffers from limitations due to the assumptions made to simplify its implementation. In particular, the protocol can block if all the members are erroneously suspected by FN, and it can slow down all of the client/server interactions if a single correct replica becomes 'slow' without being suspected by FN. These limitations are overcome by the protocol described in [25].

5. PERFORMANCE ANALYSIS

In this section we show the performance analysis we carried out on the IRL prototype. In particular, we first introduce the testbed platform and explain the set of experiments. Then we show a wide set of latency and throughput measurements that show the feasibility of the 3T approach to software replication.

5.1. Testbed platform and preliminary experiments

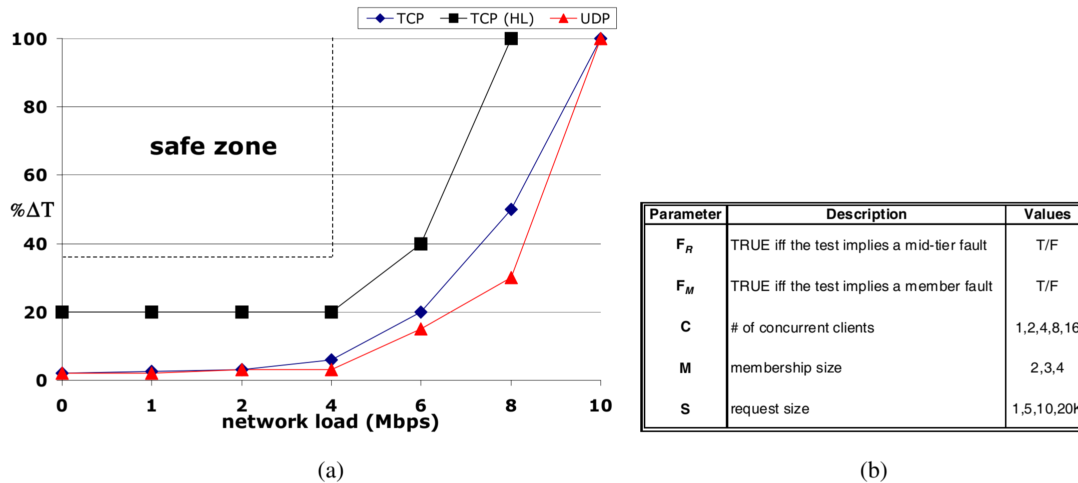
Our testbed environment consists of six Intel Pentium II 600 workstations that run the Microsoft Windows NT Workstation 4.1 sp6 operating system and are equipped with the Java Development Kit v.1.3.1. On each workstation two Java ORBs have been installed and configured: JacORB v1.3.21 [33] and IONA's ORBacus v.4.1 [34]. The workstations are interconnected by a 10 Mbit Ethernet LAN configured as a single collision domain.

As we consider perfect host failure detection on middle-tier elements, we first evaluated the (network and processor) load conditions that guarantee this behavior.

We fixed the LFD host heartbeating period and varied the network load from 0 to 100% using a traffic generator based on UDP multicast. Heartbeats were sent exploiting both UDP and TCP. Moreover, to evaluate sensitivity to processor load, we varied the processor load on the host running FN. The results are shown in Figure 5(a), which plots the minimum percentage increment (%ΔT) to apply to the LFD heartbeating period in order to obtain a *safe* FN host failure detection timeout as a function of network load. The usage of a *safe* timeout ensures perfect failure detection. As an example, with a maximum network load of 4 Mbps, having set the LFD host heartbeating period to 1 s, the value to set FN host failure detection timeout is at least 1.05 s (if the heartbeating exploits UDP or TCP on a lightly loaded

[†]Incremental updates are executed exploiting the FT-CORBA Updateable interface methods (if implemented). Otherwise the Checkpointable interface methods are exploited, performing full state transfers.

Copyright © 2003 John Wiley & Sons, Ltd.



Parameter	Description	Values
F_R	TRUE iff the test implies a mid-tier fault	T/F
F_M	TRUE iff the test implies a member fault	T/F
C	# of concurrent clients	1,2,4,8,16
M	membership size	2,3,4
S	request size	1,5,10,20K

(a) (b)

Figure 5. FN accuracy (a) and experimental parameters (b).

processor) or 1.2 s (if TCP is used on a highly loaded processor). All the experiments were carried out in the *safe zone* depicted in Figure 5(a) by controlling the timeouts and the load of the processors and network in order to avoid false suspicions. In particular, the host heartbeating period was set to 500 ms while the host heartbeating timeout was set to 1 s. Similarly, concerning object-level failure detection, we set the FaultMonitoringInterval to 500 ms and the FaultMonitoringTimeout to 1 s. In this setting, we observed that both host and object fault notifications sent by FN reach subscribers within 1.2 s from the crash event.

The parameters of the experiments are described in Figure 5(b). We measured IRL average client latency and overall system throughput[‡] by using a simple single-threaded 'hello' server that accepts requests of variable size (S) and immediately replies. The server also implements the Updatable interface, invoked by OGH replicas during the recovery protocol. However, state updates are left empty in order to not influence measurements, i.e. the GetUpdate() method invocation always returns an empty state.

To compare IRL latency and throughput with the standard CORBA performances, we first measured the average latency and throughput of a simple interaction between a client and a non-replicated server. We also varied the number of concurrent clients invoking a singleton 'hello' server instance. Results are shown in Table I.

In the following, we denote as L^* the ratio between the latency value measured in the IRL experiments and the latency of the corresponding simple client/server interaction shown in Table I.

[‡]Averages are evaluated by letting each client invoke a batch of 50 000 requests 10 times.

Table I. Basic client/server interaction benchmarks.

	Clients (C)				
	1	2	4	8	16
Client latency (ms)					
JacORB	1.28	1.37	2.30	4.34	8.30
ORBacus	1.30	1.38	2.23	3.47	7.08
Overall system throughput (requests/second)					
JacORB	769	1458	1741	1841	1926
ORBacus	729	1448	1808	2308	2262

5.2. Stateless replication performance

In this replication scheme, object group members are directly accessed by clients that embed an IRL ORGW component. Figure 6(a) shows the L^* value (on top of the bars) and the absolute latency value (inside the bars in square brackets) obtained by setting $C = 1$, $M = 2$, $S = 1$ Kb in both the failure-free ($F_M = F$) and the faulty-member ($F_M = T$) scenarios. Members and their clients were running on different hosts. Note that ORGW introduces little delay in failure-free runs (about 13% on JacORB, 31% on ORBacus) and it triplicates latency upon transparently reinvoking after a member failure. From Figure 6(b), it can be devised that Jacorb implements very thin interceptors with respect to ORBacus in non-reinvocation scenarios while ORBacus interceptors are more efficient during request redirection.

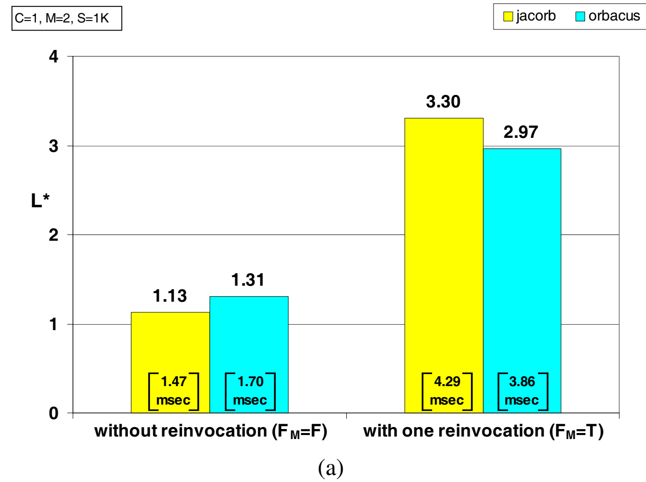
5.3. Stateful replication performance

In the following experiments we measure the performance of the 3T replication protocol described in Section 4.3. Therefore, C clients are equipped with ORGW and invoke requests of size S on the primary OGH replica that forwards them to an object group composed of M members. Each member is equipped with a co-located[§] IRGW component. In the following experiments, we vary C, M and S. We only consider *primary OGH* object failures ($F_M = F$, F_R varies) as the delay introduced by a member failure is only due to the failure detection infrastructure and can be easily measured once the host- and object-level timeouts have been set[¶]. Clients and object group members are deployed on hosts not running middle-tier replicas.

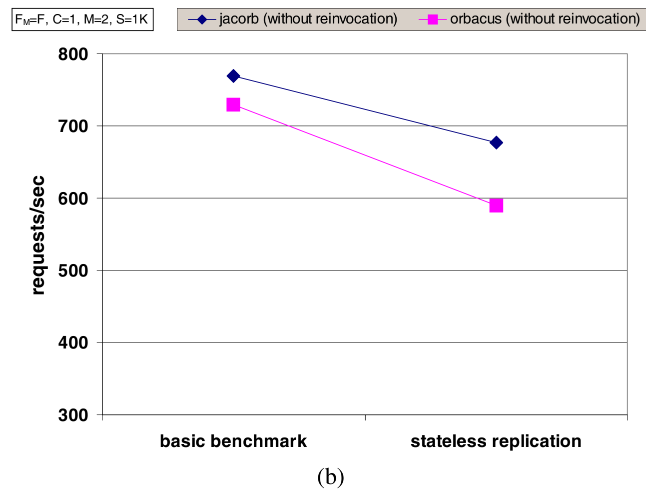
Experiment 1. In this experiment we evaluated client latency and throughput of a stateful object group as a function of the object group membership size in failure-free runs and upon a primary OGH crash. Therefore we set $F_M = F$, $C = 1$, $S = 1$ kB and vary F_R from true to false and $M \in \{2, 3, 4\}$.

[§]Two or more CORBA objects are co-located if they run within the same process [35] sharing the same addressable space.
[¶]As mentioned above, in our configuration, both object and host fault reports reach subscribers in 1.2 s.

Copyright © 2003 John Wiley & Sons, Ltd.



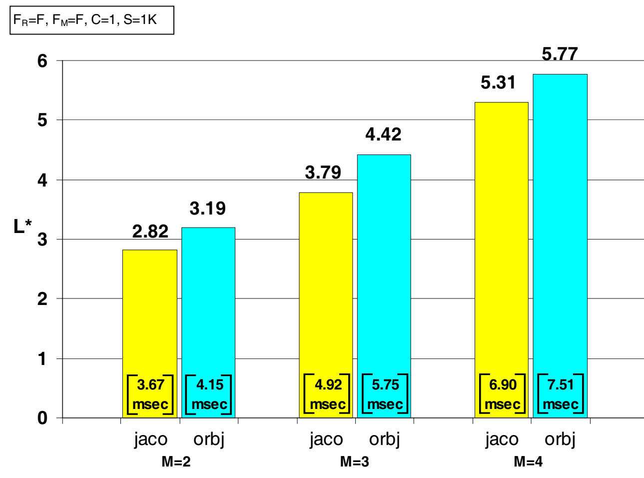
(a)



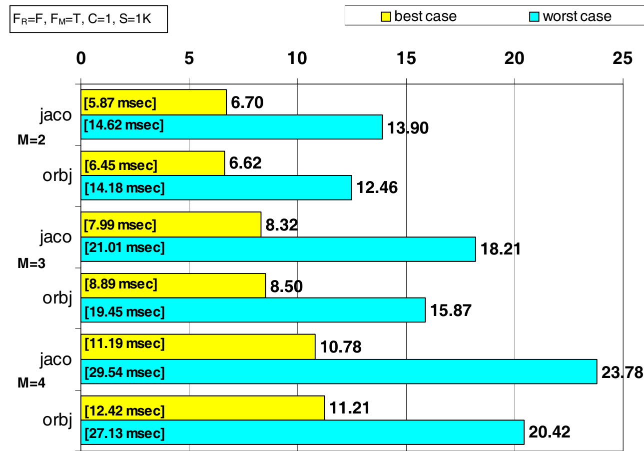
(b)

Figure 6. Stateless replication performance: (a) client latency; (b) stateless object group throughput.

Copyright © 2003 John Wiley & Sons, Ltd.

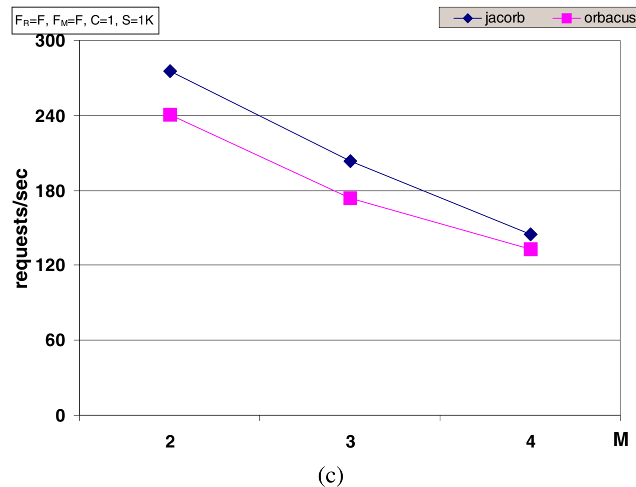


(a)



(b)

Figure 7. Stateful replication performance as a function of M: (a) client latency in failure-free runs; (b) worst and best client latency increments due to the OGH recovery protocol; (c) stateful object group throughput as a function of M in failure-free runs.



(c)

Figure 7. Continued.

Figure 7 shows the experimental results. In particular, Figure 7(a) shows the L^* ratio and absolute latency values as a function of M in the failure-free ($F_R = F$) case. Therefore, with respect to a basic benchmark (Table I, $C = 1$) and depending on the membership size (M), IRL takes about three to six times as long to complete a client interaction with a stateful object group in failure-free runs. Latency increases with M as a consequence of the OGH protocol waiting for **all** replies before starting to serve a new client request.

Figure 7(b) plots the client latency increment due to the OGH recovery protocol executed after a middle-tier failure ($F_R = T$). As the OGH recovery protocol carried out by a new primary has a best and a worst case (members are consistent after a primary OGH failure or there exists a single most updated member, respectively), we draw both the minimum and maximum delays introduced by the recovery protocol. Differences in the costs of the recovery phases between the two platforms are mainly due to JacORB not optimizing invocations between co-located objects (i.e. IRGW and the member it wraps). As a consequence, ORBacus outperforms JacORB during recovery.

Concerning throughput, Figure 7(c) shows the object group overall throughput of the stateful object group. It results that throughput reduces by roughly 70% on ORBacus and by 60% on Jacorb if $M = 2$ and by 80% on both platforms if $M = 4$ with respect to the basic benchmarks shown in Table I ($C = 1$).

Figure 8 shows the relative cost of each IRL component in a failure-free client interaction with a stateful object group, as well as the absolute time values in milliseconds taken by each component to perform its task. As expected, the time spent by a primary OGH to complete an invocation linearly increases with M. Figure 8 also confirms that JacORB portable interceptors outperform the ORBacus ones (see ORGW time), while JacORB does not optimize co-located invocations, conversely to ORBacus (see IRGW time).

Copyright © 2003 John Wiley & Sons, Ltd.

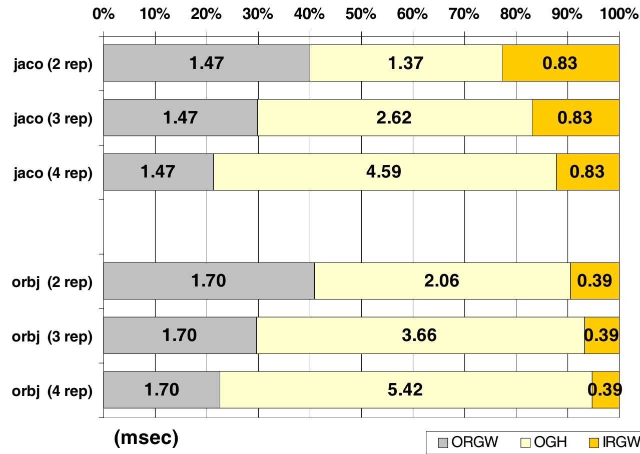


Figure 8. Relative cost of IRL components in a client/server interaction with a stateful object group.

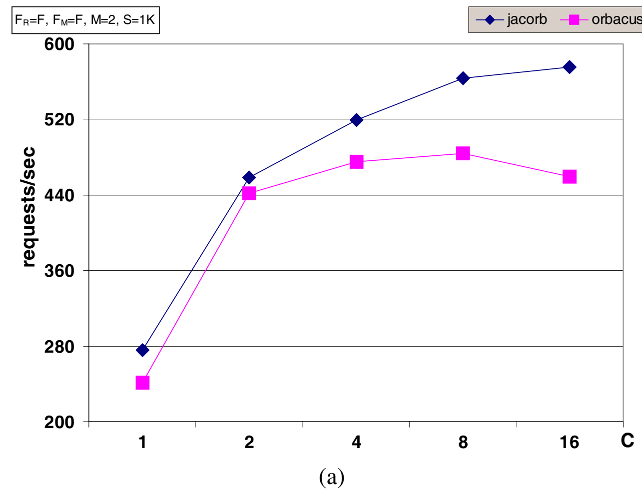
Experiment 2. In this experiment we measured the throughput of a stateful object group as a function of the number of concurrent clients. Therefore we set $F_M = F_R = F$, $M = 2$ (minimum fault tolerance), $S = 1$ kB and we vary $C \in \{2, 4, 8, 16\}$. The overall stateful object group throughput is depicted in Figure 9(a). By comparing these results with the basic benchmarks (Table I), it follows that: (i) throughput increases as long as the underlying ORB platform allows for this, then it reaches a flat; and (ii) throughput reduces to about the 35% of the throughput measured in the basic benchmarks of Table I.

Experiment 3. In this experiment we measured throughput as a function of the request size. Therefore we set $F_M = F_R = F$, $M = 2$ (minimum fault tolerance), $C = 1$ and we vary $S \in \{1, 5, 10, 20\}$ kB. The resulting plot is depicted in Figure 9(b), in which we also plot the throughput of a simple client/server interaction as a function of the client request size. As for the previous experiment, it follows that: (i) throughput decreases with the request size as the underlying ORB platform does; and (ii) the throughput still reduces to about 35% with respect to the basic benchmarks.

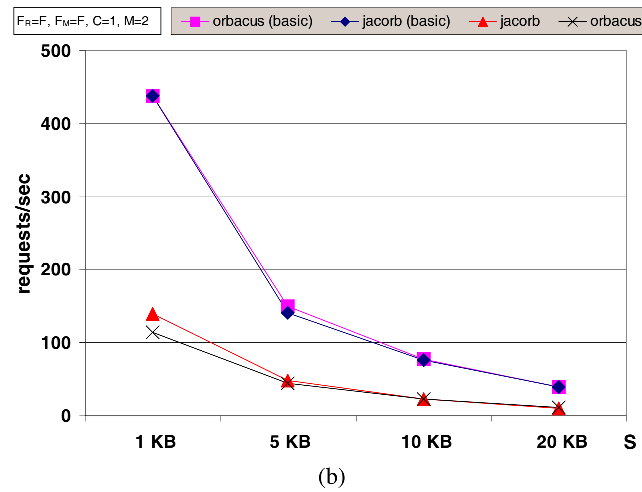
5.4. Lessons learned

Experiment 1 shows that portable interceptors are a useful and cheap tool for emulating the FT-CORBA invocation semantic without impacting on the ORB code, as they introduce a small latency increment on requests addressed to object groups in the failure-free scenario and a reasonable delay in the presence of failures. Experiments 2 and 3 mainly demonstrate that a 3T approach to replication is sound and feasible and suggest that the main limitation of the current prototype is in waiting for replies from all the replicas. To overcome this limitation, it is necessary to exchange messages among the middle-tier entities. This coupling can then be exploited to: (i) eliminate the recovery protocol and

Copyright © 2003 John Wiley & Sons, Ltd.



(a)



(b)

Figure 9. Stateful replication performance as a function of the client number C (a) and request size S (b).

Copyright © 2003 John Wiley & Sons, Ltd.

its latency increments upon failures; and (ii) weaken the assumption of perfect failure detection on middle-tier entities [25].

Concerning the overall IRL performance, it is easy to see that the first consequence of the 'above the ORB' design choice consists of binding IRL performance to the efficiency and scalability of the underlying ORB platform. This could turn out to be expensive, especially when emulating a multicast through a set of IIOP invocations performed by a middle-tier, as in the case of the current IRL prototype. On the other hand, if the accent is on integration, such a design can simplify the task of integrating heterogeneous objects while making them highly available [36]. Let us also note that improvements in the underlying ORB platform transparently turn into IRL enhancements as well, as newly available standard services, e.g. the recent unreliable multicast specification [37], can be easily embedded in the IRL framework to achieve better performances.

Concerning the prototype development, we faced some interoperability limitations, mainly due to the lack of interoperable CORBA services: we observed that CORBA services implementations are seldom non-interoperable with applications running on a different ORBs, e.g. it is not possible to exploit the ORBacus interface repository implementation from a JacORB application. To overcome this issue, we made extensive use of the facade design pattern [38]. With regard to portability, we developed a main version using ORBacus and then we ported it to JacORB. The porting phase posed interesting challenges mainly due to the differences among the 'dialects' spoken by different CORBA implementations. Indeed, it is often the case that different implementations adopt slightly different mappings of the specification to provide the same functionality. For example, we faced problems such as different initialization primitives and policies for services such as interceptors, interface repository, etc., as well as lower-level issues related to the CORBA language mapping, e.g. JacORB's CORBA `any` type mapping is different from the ORBacus one. The two platforms also differ in the classes handling CORBA system exceptions. However, DII and DSI are implemented in strict adherence to the standard and this significantly simplified the porting phase.

6. RELATED WORK

In this section we first compare our 3T architecture with other related proposals and then we deal with existing systems for CORBA fault tolerance, comparing them to IRL.

6.1. 3T systems

A 3T architecture for transparent software replication has been proposed in [39] as a lightweight approach to fault tolerance for applications that do not have strong requirements in terms of data consistency. This approach builds on the client-side mechanisms of FT-CORBA and of semantic knowledge of the server objects (provided by application developers [22]) to mediate distributed interactions in an efficient manner.

Another approach related to 3T replication has been presented in [40], in which the authors define a client–server architecture where servers run agreement protocols (e.g. group membership, atomic multicast, etc.) on behalf of clients. The idea underlying this work is to isolate entities called *servers* involved in taking system-wide consistent decisions in a fault-tolerant manner, while leaving clients to invoke such functionality upon necessity, e.g. when they need to agree on a total order of messages.

Copyright © 2003 John Wiley & Sons, Ltd.

As a consequence, only the servers must run in a partially synchronous distributed system, which is, in this case, an asynchronous system augmented with unreliable failure detectors [4].

3T systems have also gained popularity in the transactional system area. As an example, in [41] a 3T architecture is exploited to coordinate distributed transactions while enforcing an exactly-once semantic despite client reinvocations. In this scheme, the middle-tier acts as a replicated, highly available transaction manager running within an asynchronous system augmented with unreliable failure detectors to implement a non-blocking, consensus-based transaction coordination protocol. Let us note that a similar approach is suitable for implementing the fault tolerance of persistent CORBA objects using transactions and it could be easily introduced into the IRL infrastructure.

6.2. FT-CORBA systems

A large number of systems has been proposed in the last decade to address the problem of adding fault tolerance to CORBA objects. The research on this topic mainly relies on group communications as either: (1) it proposes to map CORBA requests addressed to replicated objects onto messages sent to replicas through an underlying group toolkit; or (2) it pushes up the group communication programming model into the CORBA framework providing developers with IDL interfaces to group communication services.

With respect to point (1), it is possible to further distinguish between systems on the basis of how this mapping is performed. In the so-called *intergration* approach, adopted by the Isis+Orbix, Electra [42] and AQuA [43] systems, the ORB is modified to perform the mapping. This brings advantages in terms of replication transparency and performances. However, this approach is non-interoperable and suffers from costly adaptations when ORB upgrades. In the *interception* approach, adopted by Eternal [44] and by the Fault-Tolerance Service (FTS) [45], client requests leave an unmodified ORB and are then intercepted. In Eternal, interception is implemented through an OS-dependent layer that relays group requests to the Totem [46] group toolkit, which reliably sends requests to replicas. On the replica side, another thin interception layer is in charge of passing the request up to the replica ORB, catching the result outgoing from the ORB and returning it to the client. In FTS, client request interception is achieved through standard portable interceptors which are in charge of implementing transparent request redirection and reinvocation (in a way similar to IRL) and duplicate filtering of multiple results. To maintain consistency, FTS enhances the replica object adapter which is in charge of: (i) invoking the underlying Ensemble [47] group toolkit to update other replicas; and (ii) performing logging and duplicate filtering of already executed requests. A different approach has been pursued by the $\mathcal{FRIENDS}$ system [48] that focuses on the adoption of reflection and meta-object protocols [49] to build a flexible and extensible fault-tolerance infrastructure. The resulting architecture still makes use of group communications that are transparently embedded in the application code through a specialized compiler. Eternal, FTS and $\mathcal{FRIENDS}$ do not impact on the ORB and demand the task of maintaining strong replica consistency to a layer running *under* each replica ORB. This implies ease of portability and good performance. However, replicas are tightly coupled and must run in a partially synchronous system, as asynchrony can cause the problem of system instability at the group communication level.

The previous problem has even a greater impact on systems following the *service approach*, i.e. the Object Group Service (OGS) [50] and the NewTOP object group service [51] that build a group communication toolkit exploiting the ORB as the underlying communication channel and offer developers IDL interfaces to use such services. This implies that these toolkits are fully

Copyright © 2003 John Wiley & Sons, Ltd.

portable and interoperable, but the developer must explicitly handle replica consistency though group communications. Note that in these systems the underlying ORB can introduce asynchrony. Also, Maestro [52] can be broadly considered as a service-approach system: it provides an object-oriented interface to Ensemble groups that can be accessed by CORBA clients trough IIOP. The focus in Maestro is on letting 'legacy' Ensemble applications be accessed easily by CORBA clients.

Previous analysis shows that, differently from IRL, none of the above-mentioned systems satisfies *client/server asynchrony*, as in these systems strong replica consistency is enforced by tightly coupled replicas that either implement group communications or communicate through them. On the other hand, IRL and, more generally, systems adopting a 3T architecture, actually confines the need of partial synchrony assumptions to a specific region of the system.

Furthermore, by adopting an 'above the ORB' design, IRL easily achieves interoperability and portability, sacrificing performance with respect to other systems which bypass the ORB to communicate with the replicas.

Concerning the adherence to the FT-CORBA standard, to the best of our knowledge, only the DOORS [53] and Eternal adopt compliant interfaces and only Eternal provides the full set of functionality. In particular, DOORS adopts an 'above the ORB' design to provide users with a failure detection and a checkpointing service. These services can be exploited to implement passive replication with application controlled consistency.

7. CONCLUSIONS

Maintaining *strong* replica consistency among the replicas of a stateful service deployed across an asynchronous distributed system is a real practical challenge. Commonly available architectures and tools for software replication adopt 2T techniques requiring replicas to run in a partially synchronous distributed system, i.e. they need some additional timing assumptions on the underlying system connecting the replicas. The assumptions are required to let replicas agree on the operations they perform and on their overall effects (output, state, etc.). Asynchrony can prevent these systems from working as expected. For this reason, common solutions to software replication enforcing strong replica consistency adopt workstation clusters as their underlying computing platform.

In this paper we first introduced the *client/server-asynchrony* architectural property for systems implementing software replication. An architecture satisfying this property would allow the system designer to deploy clients and replicas having no *a priori* knowledge of the execution environment or network behavior. Then we proposed a 3T architecture for software replication that frees replicas from solving agreement problems transferring these tasks to a specialized middle-tier. This allows the deployment of clients and server replicas across an asynchronous distributed system, thus satisfying the *client/server-asynchrony* property and restricting the need for partial synchrony to the region where the middle-tier runs. The main advantages and drawbacks of the 3T architecture have also been analyzed.

Finally, a FT-CORBA compliant architecture, namely IRL, has been presented, describing the techniques adopted to achieve infrastructure replication and CORBA object replication. In particular, IRL adopts a 3T architecture for software replication to provide 'infrastructure controlled consistency' of stateful CORBA objects. IRL is portable and interoperable, as it can run on several CORBA compliant ORBs while clients and server replicas lay upon standard ORBs implementing only the standard IIOP protocol. Therefore, to the best of our knowledge, IRL is the first FT-CORBA compliant

Copyright © 2003 John Wiley & Sons, Ltd.

platform that does not require all replicas to run on a single ORB. This makes IRL suitable for the dependable integration of CORBA applications running across domains heterogeneous in terms of platforms, networking infrastructure, etc. [36]. An extensive performance study of an IRL prototype has also been carried out and lessons learned have been presented. The current prototype implements a simple 3T replication protocol. Performance shows that our 3T approach to replication is feasible despite the protocol limitations.

At the time of this writing, several works stemming from the lessons learned are currently being carried out on the IRL prototype. For example, we are implementing a sequencer-based 3T replication protocol (described in [25]) that: (i) does not need failure detection on middle-tier entities and object group members to enforce liveness; (ii) allows every OGH replica to serve client requests; and (iii) returns a reply to the client as soon as the first reply is received by the middle-tier from the end-tier. We are also evaluating the use of semantic knowledge [22], in a way similar to [39], in order to enhance stateful object group performance. Finally, we are working on a fault-tolerant version of FN based on an active replication scheme, in order to let the infrastructure quickly react to object and host failures even upon FN replica crashes.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions, which greatly improved the content and presentation of this work. The paper also benefits from several discussions with Yair Amir, Roy Friedman, Dahlia Malkhi, Andre Schiper and Robbert Van Renesse. Special thanks also go to Sean Baker (Iona Technologies), Benoit Foucher (ORBacus) and Nicolas Noffke (JacORB) for the support provided during the IRL prototype development. Finally, the authors thank Paolo Papa, Alessandro Termini and Luigi Verde for their contributions to the IRL prototype and Stephen Kimani for his kind and valuable help.

This work is partially supported by a grant from Alenia Marconi Systems, a grant from MURST in the context of project 'DAQUINCIS' and by a grant of the EU in the context of the IST project MIDAS 'Middleware for Composable and Dynamically Adaptable Services'.

REFERENCES

1. Schneider FB. Replication management using the state machine approach. *Distributed Systems*, Mullender S (ed.). ACM Press/Addison-Wesley, 1993.
2. Budhiraja N, Schneider FN, Toueg S, Marzullo K. The primary-backup approach. *Distributed Systems*, Mullender S (ed.). ACM Press/Addison-Wesley, 1993; 199–216.
3. Défago X, Schiper A, Sergent N. Semi-passive replication. *Proceedings 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, West Lafayette, IN, October 1998. IEEE Computer Society Press, 1998; 43–50.
4. Chandra T, Toueg S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM* 1996; **43**(2):225–267.
5. Guerraoui R, Schiper A. Software-based replication for fault tolerance. *IEEE Computer—Special Issue on Fault Tolerance* 1997; **30**:68–74.
6. Fischer M, Lynch N, Patterson M. Impossibility of distributed consensus with one faulty process. *Journal of the ACM* 1985; **32**(2):374–382.
7. Birman KP. A review of experiences with reliable multicast. *Software—Practice and Experience* 1999; **29**(9):741–774.
8. Birman KP, Hayden M, Ozkasap O, Xiao Z, Budiu M, Minsky Y. Bimodal multicast. *ACM Transactions on Computer Systems* 1999; **17**(2):41–88.
9. Dolev D, Dwork C, Stockmeyer L. On the minimal synchronism needed for distributed consensus. *Journal of the ACM* 1987; **34**(1):77–97.
10. Amir Y, Stanton J. The spread wide area group communication system. *Technical Report CNDS-98-4*, Center for Networking and Distributed Systems, Computer Science Department, Johns Hopkins University, Baltimore, MD, April 1998.

Copyright © 2003 John Wiley & Sons, Ltd.

11. Anker T, Shnaiderman I, Dolev D. The design of Xpand: A group communication system for wide area. *Technical Report HUJI-CSE-LTR-2000-31*, The Hebrew University of Jerusalem, Computer Science Department, July 2000.
12. Dolev D, Malkhi D. The Transis approach to high availability cluster communication. *Communications of the ACM* 1996; **39**(4):64–70.
13. Object Management Group (OMG), Framingham, MA, USA. *Fault Tolerant CORBA Specification, V1.0*. OMG Document ptc/2000-12-06 edition, April 2000. OMG Final Adopted Specification.
14. Fetzer C, Cristian F. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems* 1999; **10**(6):642–657.
15. Herlihy M, Wing J. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 1990; **12**(3):463–492.
16. Felber P, Schiper A. Optimistic active replication. *Proceedings 21st International Conference on Distributed Computing Systems (ICDCS'2001)*, Phoenix, AZ. IEEE Computer Society Press, 2001.
17. Birman K, Schiper A, Stephenson P. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems* 1991; **9**(3):272–314.
18. Birman K, Joseph T. Exploiting virtual synchrony in distributed systems. *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, December 1987. ACM Press, 1987; 123–138.
19. Friedman R, Birman K. Using group communication technology to develop a reliable and scalable distributed IN coprocessor. *Proceedings of the TINA 96 Conference*, September 1996.
20. Goft G, Lotem EY. The as/400 cluster engine: A case study. *Proceedings 1999 IEEE International Workshops on Parallel Processing*, 1999. IEEE Computer Society Press, 1999; 44–49.
21. Baldoni R, Marchetti C. Software replication in three-tiers architectures: Is it a real challenge?. *Proceedings 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'2001)*, Bologna, Italy, November 2001; 133–139.
22. Felber P, Jai B, Smith M, Rastogi R. Using semantic knowledge of distributed objects to increase reliability and availability. *Proceedings 6th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS01)*, Rome, Italy, January 2001. IEEE Computer Society Press, 2001; 153–160.
23. Object Management Group (OMG) *The Common Object Request Broker Architecture and Specifications. Revision 2.4.2*, OMG Document formal edition, Framingham, MA, February 2001. OMG Final Adopted Specification.
24. IRL Project Web site. http://www.dis.uniroma1.it/∼irl.
25. Baldoni R, Marchetti C, Tucci-Piergiovanni S. Asynchronous active replication in three-tier distributed systems. *Proceedings 9th IEEE Pacific Rim Symposium on Dependable Computing (PRDC02)*, Tsukuba, Japan, December 2002. IEEE Computer Society Press, 2002; 19–26.
26. Baldoni R, Marchetti C, Verde L. CORBA request portable interceptors: Analysis and applications. *Concurrency and Computation: Practice & Experience*. Wiley. To appear.
27. Object Management Group (OMG). *Portable Interceptor Specification*. OMG Document orbos edition, Framingham, MA, December 1999. OMG Final Adopted Specification.
28. Birman K. The process group approach to reliable distributed computing. *Communications of the ACM* 1993; **36**(12): 37–53.
29. Baldoni R, Marchetti C, Panella R, Verde L. Handling FT-CORBA compliant interoperable object group reference. *Proceedings 6th IEEE International Workshop on Object Oriented Real-time Dependable Systems (WORDS'02)*, Santa Barbara, CA, January 2002. IEEE Computer Society Press, 2002; 37–44.
30. Babaoglu O, Davoli R, Montresor A. Group communication in partitionable systems: Specification and algorithms. *IEEE Transactions on Software Engineering* 2001; **27**(4):308–336.
31. Fetzer C. Enforcing perfect failure detection. *Proceedings 21st International Conference on Distributed Systems*, Phoenix, AZ, April 2001. IEEE Computer Society Press, 2001.
32. Baldoni R, Marchetti C, Termini A. Active software replication through a three-tier approach. *Proceedings 22nd IEEE International Symposium on Reliable Distributed Systems (SRDS02)*, Osaka, Japan, October 2002; 109–118.
33. JacORB Web Site. http://www.jacorb.org.
34. IONA Web Site. http://www.iona.com.
35. Henning M, Vinoski S. *Advanced CORBA Programming with C++*. Addison-Wesley/Longman, 1999.
36. Marchetti C, Virgillito A, Baldoni R. Enhancing availability of cooperative applications through interoperable middleware. *Journal of Information Science and Engineering*; **19**(1):39–58.
37. Object Management Group (OMG). *Unreliable Multicast*, OMG Document ptc/2001-11-08 edition, Framingham, MA, October 2001. OMG Draft Adopted Specification.
38. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
39. Felber P. Lightweight fault tolerance in CORBA. *Proceedings International Symposium on Distributed Objects and Applications (DOA'01)*, Rome, Italy, September 2001. IEEE Computer Society Press, 2001; 239–247.
40. Guerraoui R, Schiper A. The generic consensus service. *IEEE Transactions on Software Engineering* 2001; **27**(1):29–41.

Copyright © 2003 John Wiley & Sons, Ltd.

41. Guerraoui R, Frølund S. Implementing e-transactions with asynchronous replication. *IEEE Transactions on Parallel and Distributed Systems* 2001; **12**(2):133–146.
42. Landis S, Maffeis S. Building reliable distributed systems with CORBA. *Theory and Practice of Object Systems* 1997; **3**(1):31–43.
43. Cukier M, Ren J, Sabnis C, Henke D, Pistole J, Sanders WH, Bakken DE, Berman, ME, Karr DA, Schantz RE. AQuA: An adaptive architecture that provides dependable distributed objects. *Proceedings 17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, IN, October 1998. IEEE Computer Society Press, 1998; 245–253.
44. Moser LE, Melliar-Smith PM, Narasimhan P, Tewksbury LA, Kalogeraki V. The Eternal system: An architecture for enterprise applications. *Proceedings 3rd International Enterprise Distributed Object Computing Conference (EDOC'99)*, Mannheim, Germany, July 1999. IEEE Computer Society Press, 1999; 214–222.
45. Friedman R, Hadad E. FTS: A high-performance CORBA fault-tolerance service. *Proceedings 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, 2002; 61–68.
46. Moser LE, Melliar-Smith PM, Agarwal DA, Budhia RK, Lingley-Papadopoulos CA, Archambault TP. The Totem system. *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, Pasadena, CA, 1995; 61–66.
47. Hayden MG. The Ensemble system. *PhD Thesis*, Cornell University, Ithaca, NY 1998.
48. Killijian M-O, Fabre JC. Implementing a reflective fully-tolerant CORBA system. *Proceedings 19th IEEE Symposium on Reliable Distributed Systems (SRDS-2000)*. IEEE Computer Society Press, 2000; 154–163.
49. Fabre J-C, Perennou T. A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers, Special Issue on Dependability of Computing Systems* 1998; **47**(1):78–95.
50. Felber PA, Guerraoui R. The implementation of a CORBA group communication service. *Theory and Practice of Object Systems* 1998; **4**(2):93–105.
51. Morgan G, Shrivastava SK, Ezhilchelvan PD, Little MC. Design and implementation of a CORBA fault-tolerant object group service. *Proceedings of the 2nd IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*. Kluwer, 1999; 179–185.
52. Birman AV, Birman KP. The Maestro approach to building reliable interoperable distributed applications with multiple execution styles. *Theory and Practice of Object Systems* 1998; **4**(2):73–80.
53. Chung P, Huang Y, Yajnik S, Liang D, Shih J. DOORS—providing fault tolerance to CORBA objects. *Poster Session at IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England, 1998. Springer, 1998.

Copyright © 2003 John Wiley & Sons, Ltd.