IRL: A Three-Tier Approach to FT-CORBA Infrastructures

R. Baldoni, C. Marchetti and A. Termini
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, 00198, Roma, Italy
email: {baldoni,marchet,termini}@dis.uniroma1.it

Abstract

A replication logic is the set of protocols and mechanisms implementing a software replication technique. A three-tier approach to replication consists in separating the replication logic from both clients and servers by embedding such logic in a mid-tier. This novel approach allows (i) to design thin client (embedding the presentation code and some redirection/retransmission mechanism), (ii) to let servers replicas (the end-tier) be accessed by common request/response asynchronous invocation, and then (iii) to deploy server replicas on an asynchronous distributed system without the burden of managing complex distributed protocols (this improves scalability and simplifies replica management). In this paper we also present the Interoperable Replication Logic (IRL) architecture which is a Fault Tolerant CORBA compliant infrastructure exploiting a three-tier approach to replicate CORBA objects. We finally present an extensive performance study of an IRL prototype.

1 Introduction

Active [24], passive [5], semi-passive [7] and quorum replication [18] are well-known approaches to build fault-tolerant services in distributed systems by using software replication. These approaches are inherently two-tiers in the sense that independent clients (client-tier) interact with a set of server replicas (the end-tier) which implement the fault-tolerant service.

In these approaches server replicas implement both the service they provide as well as mechanisms, protocols and data structures necessary to handle a replication scheme (i.e., the replication logic). As examples, in passive replication each replica has to monitor the state (correct/failed) of the primary and to maintain an agreement on the membership of the group composed by server replicas. In active replication, server replicas must be equipped with a total order multicast primitive that hides a consensus protocol run among replicas to let them agree on the order of message deliveries. These mechanisms impose tight coupling among server replicas that leads such replicas (i) to explicitly know each other, (ii) to communicate directly and (iii) to play an active role in handling their replication.

The basic idea behind our study is to decouple the aspects related to replication from the actual service implementation by encapsulating the replication logic in a software tier external to the service. From an operational viewpoint, such software tier undertakes the burden of dealing with the service replication, i.e. of maintaining consistency among the server replicas. As a consequence the resulting architecture is three-tier, where the replication logic acts as a mid-tier between clients and server replicas. Clients send requests to the mid-tier that forwards them to the end-tier. Note that in this scenario, clients are expected to be thin (embedding at most a retransmission/redirection mechanism) and each server replica communicates only with the mid-tier on a point-to-point basis following a simple request/response message pattern. A server replica receives requests from the mid-tier, processes them and return replies. In this sense replicas play a passive role in handling replication. This simple message pattern allows to disseminate replicas across the Internet while keeping the entities managing the replication logic in a system with a higher level of synchrony, e.g. a LAN. From the end-tier point of view, this makes the replication much more flexible and lightweight than previous schemes.

In the remainder of this paper we first give an overview of three-tier replication (Section 2), then we introduce the Interoperable Replication Logic (IRL) design, which is a CORBA infrastructure compliant with the recent FT-CORBA specification [22] and adopting a three-tier approach to replication (Section 3). In Section 4 we present the IRL prototype that has been developed, tested and evaluated in our department. Hence, Section 5 shows a comprehensive performance analysis carried out on such prototype. Finally, Section 6 concludes the paper.

2 Three-tier Replication

Distributed architectures have been usually built as two-tiers: clients and servers. Clients embed the presentation part of an application, servers embed application data and the application logic is usually split between clients and servers. In recent years, increasing requirements of software modularity pushed application logic in a specific software tier. The result are three-tier software architectures, in which the client code only concerns the presentation logic, the mid-tier embeds the application logic possibly without containing any part of the application state, which should be completely managed by *independent* servers accessed by standard interfaces ([12]).

The basic idea behind this paper is to embed the replication logic (i.e., the the set of protocols, mechanisms and data structures that allow to implement a software replication technique by enforcing *linearizability* [14] on executions of a set of server replicas.) within a mid-tier, actually getting an three-tier replication architecture.

Three-tier replication decouples the replication logic from the functionality provided by the server replicas. In other words, it separates the protocols and data structures handling consistency of stateful server replicas from the servers themselves. This allows to design services almost independently from replication-related issues, to simplify replica deployment, and to simplify replication of services not designed having high availability in mind. Figure 1 illustrates the basic architecture.

In this basic architecture, clients are very thin entities that implement only a simple retransmision mechanism in order to cope with mid-tier failures, while replicas can at most implement simple functionality (e.g. request logging and duplicate filtering) other than the ones declared in their interface. Note that server replicas are *independent*, i.e. they do not

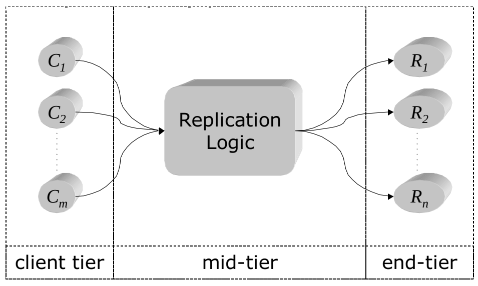


Figure 1: A Basic Architecture for Three-tier Replication

exchange messages among them. They only exchange messages with the replication logic on a point-to-point basis and following a simple request/reponse message pattern[1]. Therefore a replica is not aware of the existence of other replicas and it is also not aware of the replication scheme (e.g. active, passive or semi-passive replication) used by the replication logic to manage end-tier replicas [2].

It is easy to see that the availability of the end-tier replica functionality passes through the availability of the mid-tier. Therefore, even the mid-tier has to be fault tolerant in order to avoid single points of failure. This implies that, in the most general case, the replication logic implements both a *mid-tier replication* and an *end-tier replication* protocol.

The mid-tier replication protocol involves only mid-tier components and is used to maintain consistency on information about client/server interactions. In this way if a mid-tier component fails while carrying out a client/server interaction, another mid-tier component has to take over and conclude the job in a correct way, with respect to end-tier execution linearizability.

During a failure-free client/server interaction, the end-tier replication protocol involves (at least) a mid-tier element (the one that carries out the client/server interaction) and end-tier replicas. This protocol is based on a simple request/response message pattern in which the mid-tier component issues a set of requests, one for each replica, each replica processes the request and replies to the component. As replicas play a passive role in this interaction, the end-tier protocol can be easily implemented in asynchronous distributed systems, thus allowing for example to disseminate server replicas across the Internet. On the other hand, the mid-tier protocol could be implemented in other distributed system contexts (e.g. LANs,

[1]This enables this basic architecture to work with technologies based on TCP as well as on IIOP and SOAP (e.g. CORBA, Microsoft .NET etc).

workstation clusters etc.) and benefit of computation models with higher degrees of synchrony (e.g. partially synchronous distributed systems [6], timed asynchronous distributed systems [11]) leading to a simplification of the mid-tier protocol itself.

This flexibility and the fact that server replicas are completely independent (allowing for example simplified dynamic changes in end-tier membership) represent the main advantages of three-tier replication with respect to classical (two-tiers) replication. The price to pay is one additional hop in the client/server request invocation path with respect to classical two-tier replication.

Let us final remark that three-tier replication is not an alternative to classical two-tiers software replication, e.g. [5, 7, 10, 13, 18, 24] . It is rather an architectural solution when facing software replication in asynchronous distributed systems. Three-tier replication actually needs two-tier software replication to make the replication logic resilient to failures. The result is an architecture in which (i) asynchronous aspects are handled by a simple request/response point-to-point protocol between the mid-tier and the server replicas and (ii) two-tier replication is confined in a well-defined "box" that can even adopt, for example, a group communication toolkit to manage the mid-tier replication protocol. The selection of the mid-tier software replication technique mainly depends on the degree of synchrony of the distributed system underlying the box, which actually influences the complexity of the mid-tier protocol, and on the expected system performance with respect, for example, to failure-reaction and response times.

3 IRL FT-CORBA Compliant Design

In this section we first summarize the FT-CORBA specification and then we present IRL FT-CORBA compliant design.

3.1 Overview of FT-CORBA Specification

The FT-CORBA standard addresses the problem of increasing the availability of *deterministic* CORBA objects failing by crashing. FT-CORBA achieves fault tolerance through CORBA object redundancy, fault detection and recovery. Replicas of a CORBA object are deployed on different hosts of a *fault tolerance domain*[2] (FT-domain). Replicas of an object 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 object ([4]). If the replicated object maintains an internal state (i.e. it is *stateful*), strong replica consistency[3] has to be enforced among the states of the object group members.

To identify and address object groups, FT-CORBA introduces *Interoperable Object Group References* (IOGR*s*). Roughly speaking, an IOGR is a CORBA Interoperable Object Reference (IOR, [23]) composed by multiple *profiles*, each profile pointing either (i) to an object group member (in the cases of passive and stateless replication) or (ii) to a gateway orchestrating accesses to the object group members (in the case of active replication). IOGRs are used by FT-CORBA compliant client ORBs in order to allow client applications to uniformly access stateless and stateful object groups while benefiting of replication and failure transparency. This is achieved by modifying client ORBs to implement the *transparent client reinvocation* and *redirection* mechanisms[4].

On the server side, FT-CORBA extends the CORBA standard and the server ORBs with new mechanisms and architectural components organized into the three main features, namely *replication management*, *fault management* and *recovery management*.

Replication management consists in the creation and management of object groups and of object group members. The **ReplicationManager** component is responsible for carrying out such activities. In particular, when requested for object group creation, **ReplicationManager** exploits *local factories*[5] to create the members, collects the members' references and returns the object group reference. For stateful object groups, a replication technique (e.g. active or passive) can be selected. It is also possible to specify whether its up to the application or to the infrastructure to maintain the group membership information and the consistency among the replicated states, i.e. **ReplicationManager** can be requested to create application/infrastructure controlled membership/consistency object groups. Finally,

[2]A fault tolerance domain is collection of hosts interconnected by a non partitionable computer network.

[3]Formally, strong replica consistency of a deterministic object consists in the *linearizability* [14] of the request execution histories of the replicated object.

[4]In [3], we provide a solution based on CORBA Portable Interceptors ([21]) that lets non FT-CORBA compliant client ORBs benefit of failure and replication transparency.

[5]Local factories are provided by application developers. A distinct local factory has to be developed for each distinct object type and deployed on each host of the FT-domain.

ReplicationManager can be requested to monitor that a minimum number of members is alive inside a given object group and to perform recovery actions after object group members' failure in order to maintain such minimum number of members.

Fault Management concerns the detection of object group members' failures, the creation and the notification of fault reports and the fault report analysis. These activities are respectively carried out by the **FaultDetectors**, **FaultNotifier** and **FaultAnalyzer** components. A replicated object can be monitored for failures by a **FaultDetector** if it implements the PullMonitorable interface, i.e. it is *monitorable.* **FaultNotifier** lets clients request to monitor monitorable objects and subscribe for receiving object fault reports. Hence, **FaultNotifier** receives object fault reports from the **FaultDetectors** and propagates fault notifications to the **ReplicationManager** and other subscribed clients[6].

Recovery Management is based on two *mechanisms*, i.e. **logging** and **recovery** that exploit two IDL interfaces (Checkpointable and Updateable) that *recoverable* application objects implement to let **logging** and **recovery** mechanisms read and write their internal state. More specifically, the **logging** mechanism periodically stores on a log information (i.e. state, state update, served requests, generated replies) of recoverable objects while the **recovery** mechanism retrieves this information from the log when, for example, spawning a new member to consistently set its initial internal state.

In a FT-domain, one logical instance of the **ReplicationManager** and one of the **FaultNotifier** components must be created.

To get compliance, a subset of aforementioned features must be implemented. Different approaches can be adopted, e.g. by exploiting either a group communication toolkit (as in the case of the Eternal System [19]) or state-logging tools (as in the case of the DOORS System [20]). No matter of how it is implemented, the software infrastructure spread over the FT-domain which handles object groups and provides interfaces for their management is commonly referred to as a *fault tolerance infrastructure*(FT-infrastructure).

[6]**FaultAnalyzer** receives all the fault reports from the **FaultNotifier** and generates condensed fault reports. However, it is an *optional* component that we do not consider in this paper.

3.2 IRL Architectural Overview

IRL FT-infrastructure is made up of a set of components implemented as standard CORBA objects. The main idea underlying IRL design is to use three-tier replication in order (i) to allow independent stateful CORBA objects to be replicated and managed, (ii) to decouple the replication logic from the replicated CORBA object functionality and (iii) to let clients and CORBA objects exploit the standard CORBA Internet Inter ORB Protocol (IIOP, [23]) to interact with the mid-tier in order to get interoperability. Figure 2 illustrates the main IRL FT-infrastructure components.

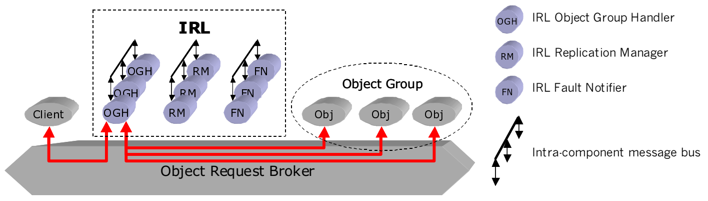


Figure 2: IRL Architecture

As the figure shows, clients of a stateful object group interact with a replicated mid-tier (IRL OGH component) through standard CORBA invocations as well as the mid-tier does with the object group members[7]. Therefore IRL results *interoperable*, as IRL components and replicated objects managed by IRL can run on ORBs from distinct vendors. This allows to overcome the FT-CORBA vendor dependence limitation [22].

Figure 2 also shows the IRL RM (ReplicationManager) and FN (FaultNotifier) components that implement standard FT-CORBA interfaces. OGH, RM and FN interact through standard CORBA invocations. However, they *can* also implement an *intra-component message bus*, which component replicas exploits in order to synchronize their internal state, i.e. maintain replica consistency. To preserve infrastructure portability, we allow intra-component message bus implementations exploiting technologies different from IIOP as far as they lay upon the underlying TCP/IP protocol stack, upon which also IIOP lays.

In the following, we present a short description of IRL components. The description is functional, in the sense that it is independent from the replication techniques adopted to replicate IRL components, which will be summarized in Section 4. Interested readers can

[7]For consistency with the FT-CORBA specification, hereafter we refer to end-tier server replicas as *object group members* and to IRL component replicas as *replicas*.

refer to [16] for further details.

IRL Object Group Handler (OGH). An IRL Object Group Handler component is associated with each stateful object group. OGH stores the IORs of the members and the information about their availability. OGH is mainly responsible for enforcing strong replica consistency among its group members' states. In particular, OGH acts as the mid-tier of a three-tier replication protocol. By exploiting IOGRS and CORBA DII and DSI mechanisms, it 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 finally returns one of them to the client (see Section 4.3)[8].

IRL Replication Manager (RM). This component is a FT-CORBA compliant ReplicationManager. 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. Note that in the case of infrastructure controlled consistency, RM also spawns a replicated mid-tier (composed by a fault-tolerant IRL OGH component) and the object group reference returned by RM points to such a mid-tier. IRL RM allows the management of stateless and statefull object groups with application/infrastructure controlled membership and consistency.

IRL Fault Notifier (FN). As FT-CORBA compliant FaultNotifier, the IRL Fault Notifier receives object fault reports from **IRL Local Failure Detectors (LFDs)**, i.e. FT-CORBA compliant FaultDetectors and subscriptions for failure notifications by clients. Upon receiving an object fault report from a LFD, FN forwards it to RM and to clients that subscribed for the faulty object. In addition to this, FN implements host failure detection by receiving heartbeats from LFDs. Upon not receiving an heartbeat within a predefined timeout value, FN creates an host fault-report that pushes to RM as well as to every client that subscribed for objects running on the faulty host.

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

[8]IRL also support stateless replication. In this case, no OGH is interposed between clients and object group members. According to FT-CORBA specification, a client simply uses IOGR profiles to directly connect to a server replica.

4 The IRL Prototype

The current IRL prototype is written in Java and runs on three different Java-CORBA platforms: JacORB freeware platform [17] and IONA's [15] ORBacus and Orbix 2000 for Java. Note that IRL design can be implemented is different ways, which mainly differ for (i) the mid-tier and end-tier replication protocols (see Section 2) embedded in the OGH component and (ii) for the techniques and technology adopted to implement the intra-component message bus, i.e. to achieve infrastructure fault tolerance. Therefore in this section we introduce the techniques adopted in the current IRL prototype in order not to introduce single points of failure in the FT-infrastructure[9]. The choice of the replication technique for each IRL component is based on the state it maintains (if any) and on its deployment and fault-tolerance requirements. To simplify the explanation, we identify two classes of components, Host-Specific IRL Components (LFD and IRL Factory) and Domain-Specific IRL Components (OGH, RM and FN).

4.1 Host-Specific IRL Components.

LFD and IRL Factory components are installed on each host of the FT-domain and their activities are related only to the host they run on (i.e. LFD monitors objects running on *its* host, IRL Factory creates objects on *its* host). As a consequence, host-specific components do not need to survive to their host crash. However, being subject to software failures, they are replicated and their replication is efficiently performed on a local basis (i.e. with no message exchange among different hosts). In particular, IRL Factory is stateless and it is simply replicated on the host it runs on by launching two separate instances that exchange *I'm alive* messages to recreate the partner upon a replica failure. On the other hand, LFD maintains a simple state composed by the references of the monitorable object running on its host. For this reason, it is replicated using cold passive replication and monitored for failures by IRL Factory, that starts a new LFD replica upon detecting a failure.

[9]Here we give just an overview of the techniques, for further details see [16].

4.2 Domain-Specific IRL Components.

IRL RM, FN and OGH components implement functionality that must survive to host failures. Therefore, domain specific components are replicated with replicas running in a separate process of distinct hosts of the FT-domain. The replication of OGH will be explained in the next subsection while the current prototype does not support FN replication[10]. So here we sketch the basic idea underlying RM replication. RM can execute outgoing invocations to local factories upon receiving a request. This makes RM nondeterministic, and then its replication necessarily follows a passive scheme. To achieve this, a singleton RM instance implementing the Checkpointable and Updateable interfaces is wrapped by a Wrapper CORBA object. Wrapper implements the intra-component message bus exploiting the Jgroup toolkit [1]. Note that the outgoing invocations executable by RM (e.g. create_object, delete_object, invoked to create and delete object groups) can be executed multiple times (e.g. upon receiving a request reinvocation due to a previous primary RM failure) without affecting the overall system external behaviour. This semantic knowledge allowed us to simplify RM replication, without dealing with duplicate filtering of requests outgoing from RM. Additional details about RM replication can be found in [16].

4.3 The IRL Prototype Three-tier Replication Protocol

As pointed out before, interactions of clients with a *stateful* object group are mediated by the IRL OGH component associated with the group. In the current IRL prototype, OGH implements passive replication as mid-tier replication protocol and active replication as end-tier replication protocol (see Section 2). We support *static* groups of both OGH and object group members[11]. Object group members have to be deterministic and can not perform outgoing invocations.

The mid-tier passive replication protocol implemented by OGH is based on perfect failure detection [6], i.e. FN does not make mistakes when detecting host crashes[12]. This assumption allows to simplify the mid-tier replication protocol implemented by OGH.

[10]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.

[11]A group is *static* when its initial membership changes (reduces) over the time only because of member failures.

[12]We experimentally evaluate the conditions under which this assumption is verified in Section 5.

Deployment and Initialization.

Client-tier. In order to let client applications benefit of transparent client reinvocation even on non FT-CORBA compliant client ORBs, client applications are augmented with the IRL Object Request Gateway (ORGW) component [3]. In short, ORGW is a CORBA Client Request Portable Interceptor [21] that (i) intercept requests addressed to object groups (i.e. using an IOGR), (ii) uniquely identifies them as the FT-CORBA standard prescribes and (iii) iteratively tries to send the request to a correct member, until either it receives a reply (that it returns to the client application) or it has tried all of the IOGR profiles without receiving a reply[13].

End-tier. Each stateful object group member is transparently wrapped by the IRL Incoming Request Gateway Component (IRGW) that implements a basic FT-CORBA logging mechanism. In short, IRGW adopts the same interface (by exploiting the Dynamic Skeleton Interface and the Interface Repository) and receives all the requests of the member it wraps. Upon receiving a request, IRGW first checks if the request is a reinvocation (exploiting the FT-CORBA compliant unique request identifier). If it is the case, IRGW returns the result that it has previously logged. Otherwise it (i) forwards the request to the member, (ii) waits until a result is produced, (iii) logs the request/reply pair and (iv) it finally returns the result to the client. To perform garbage collection of outdated request/reply pairs, IRGW exploits the FT-CORBA request expiration time contained in the unique request identifier.

In order to let OGH perform state synchronization, we assume object group members to implement at least the Checkpointable and optionally the Updateable FT-CORBA interfaces.

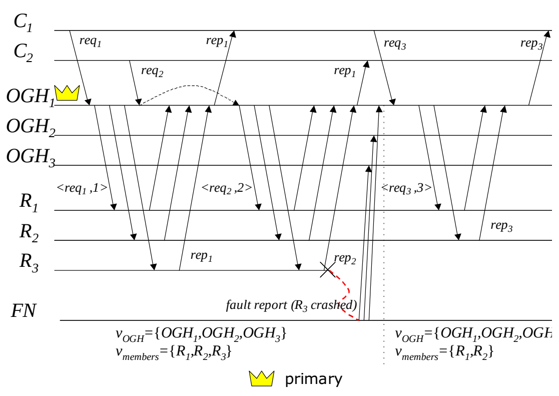
Mid-Tier. When IRL RM creates a new stateful object group, it starts a set of OGH replica (each running on a distinct host), other than object group members. Each OGH replica reads the interface of its object group member type from the CORBA Interface Repository in order to parse incoming requests on behalf of its object group members by exploiting a Dynamic Skeleton Interface. Moreover, each replica receives from IRL RM two initial views, i.e. a view containing the identifiers of the OGH replicas (v_{OGH}) and the view containing the identifiers of the object group members ($v_{members}$). Views are dynamically updated by OGH upon receiving object and host fault reports from IRL FN, to which each OGH subscribes as

[13]In this case, ORGW throws a NO_RESPONSE exception to the client application, in strict compliance of the FT-CORBA standard.

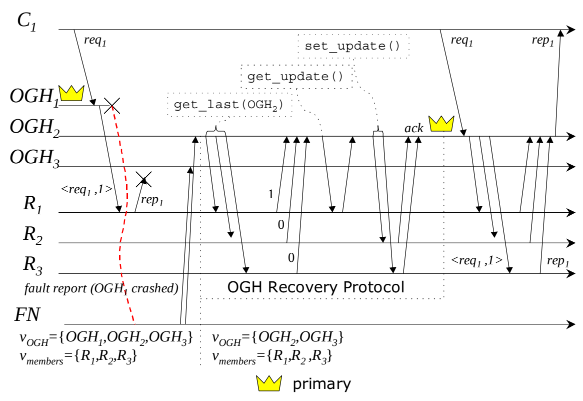
consumer for the objects contained in $v_{OGH} \cup v_{members}$.

The Protocol.

The IRL three-tier prototype protocol is illustrated in Figure 3. In Scenario 1 (Figure 3(a)) client C_1 issues request req_1 that reaches the primary OGH_1. Upon receiving the request, OGH_1 piggybacks onto req_1 a local sequence number (increased for each served request) and forwards the request to every member $R_i \in v_{members}$. Each IRGW wrapping a member, upon receiving a request stores the sequence number and then forwards the request to its replica. Then it waits for the result, logs the request and the result and returns the reply to OGH_1. Once OGH_1 has received the results from all $R_i \in v_{members}$, it returns the reply to the client. Note that if OGH_1 receives another request req_2 before completing a request processing (e.g. req_1), then req_2 is queued until rep_1 is sent to C_1. This preserves request ordering at object group members in absence of primary failures. When a member crashes (e.g. R_3 in Figure 3(a)), FN forwards a fault report to every $OGH \in v_{OGH}$, allowing the primary OGH not to undefinitevely block by waiting a reply from a crashed member. Scenario 2 (Figure 3(a))



(a) Scenario 1



(b) Scenario 2

Figure 3: The IRL Prototype Three-tier Replication Protocol

illustrates how a primary OGH crash is handled[14]. When OGH_1 crashes, FN notifies each backup of the fault. Hence, each OGH updates its local copy of v_{OGH} and decides if it is the new primary. In Scenario 2, OGH_2 is the new primary as its id appears as the first element of v_{OGH}, whose elements are ordered basing on replica identifiers. As a primary can fail during

[14]Backup OGH crashes are simply notified by FN to every OGH that updates v_{OGH}.

the processing of a request without updating all the members of v_{member} (e.g. OGH_1), OGH_2 performs a recovery protocol before starting to serve client requests. Recovery is needed to ensure update atomicity on the members of $v_{members}$ after primary failures. To achieve this, OGH_2 first determines if all the members are in the same state by invoking the IRGW get_last method[15]. This method takes as input parameter the new primary identifier[16] and returns the sequence number of the last request received by IRGW. If all the members return the same sequence number, then they are in a consistent state and OGH_2 starts serving client requests. Conversely, i.e. if some member executed an update not executed by some other members, OGH_2 gets a state update from one of the most updated members and set the update to the least updated ones. Incremental updates are executed exploiting the FT-CORBA Updateable interface methods (if implemented). Otherwise the Checkpointable interface methods are exploited, performing full state transfers. Then OGH_2 starts serving client requests as the new primary, having ensured update atomicity on object group members. As clients implement request retransmission (e.g. by ORGW), C_1 reinvokes req_1 onto OGH_2. In this situation, members are already updated and then the IRGW wrapping each member returns the logged result without repeating the member invocation and preserving the CORBA at-most-once request invocation semantic.

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 the set of experiments we carried out. Then we show a wide set of latency and throughput measurements that show the feasibility of the IRL approach to software replication.

5.1 Testbed Platform and Preliminary Experiments

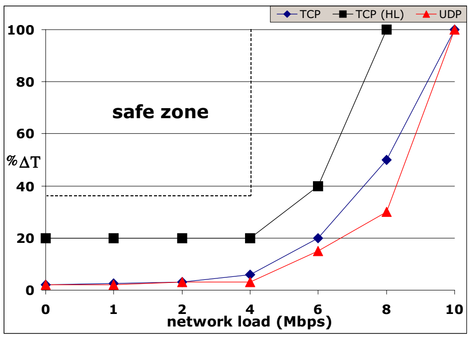
Our testbed environment consists in six Intel Pentium II 600 workstations that run Microsoft Windows NT Workstation 4.1 sp6 as operative system and are equipped with the Java Development Kit v.1.3.1. On each workstation two Java ORBs have been installed and configured:

[15]Actually, this invocations would be not necessary if the FT-CORBA get_update (get_state) method would return an update (state) sequence number along with their current result.

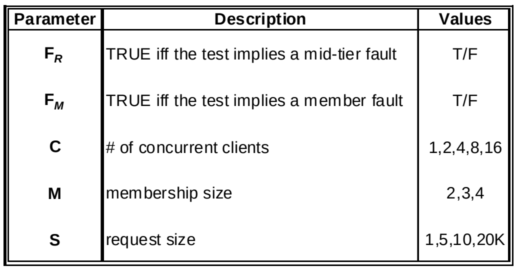
[16]The new primary identifier allows IRGW to filter out outdated request coming from crashed primaries.

JacORB v1.3.21 [17] and IONA's ORBacus v.4.1 [15][17]. The workstation are interconnected by a 10Mbit Ethernet LAN configured as a single collision domain.

As we assume perfect host failure detection on mid-tier elements (OGH), we first evaluated the system conditions that actually makes FN behave perfectly. Therefore we fixed the LFD host heartbeating period and varied the network load from 0% up to 100% using an UDP multicast based traffic generator. Heartbeats have been sent exploiting both UDP multicast and TCP. Moreover, to evaluate sensibility to processor load, we varied the processor load on the host running the primary FN replica. Results are shown in Figure 4(a), which plots the minimum percentage increment (%ΔT) to apply to LFD heartbeating period in order to obtain a *safe* FN host failure detection timeout as a function of network load. A *safe* FN host failure detection timeout allows FN to avoid false suspicions. As examples, with a maximum network load of 4Mbit/sec, having set LFD host heartbeating period to 1sec., the value to set FN host failure detection timeout is roughly either 1,05 sec. if heartbeating exploits UDP or TCP on a lightly loaded processor and 1,2 sec. if TCP/IP is used on a highly loaded processor.



(a) FN Accuracy



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

(b) Experimental Parameters

Figure 4: FN Accuracy and Experimental Parameters

All the experiments were carried out in the safe zone depicted in Figure 4(a).

In the experiment suite, we let vary the parameters described in Figure 4(b) and measure IRL average client latency and system throughput[18] by using a simple single-threaded "hello"

[17]We plan to run our test suite also on the IONA's Orbix 2000 for Java ORB and to include the results in a revised version of this paper.

[18]Averages are evaluated by letting each client invoke 10 times a batch of 50.000 requests.

server that accepts requests of variable size (S) and immediately replies[19].

IRL Latency and throughput must be compared with the results of the basic benchmarks we took, i.e. the average latency and throughput of a simple client server interaction with no fault tolerance. We vary the number of concurrent clients invoking on a singleton hello server instance. Results for the two platforms are shown in Table 1. In the following, we

Client Latency (msec)

Clients (C)	1	2	4	8	16
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/sec)

Clients (C)	1	2	4	8	16
JacORB	769	1458	1741	1841	1926
ORBacus	729	1448	1808	2308	2262

Table 1: Basic Benchmarks

report results of latency measurement as the ratio between the latency value measured in the experiments and the corresponding simple client/server interaction latency value. We refer to this ratio as L^*.

5.2 Stateless Replication Performance

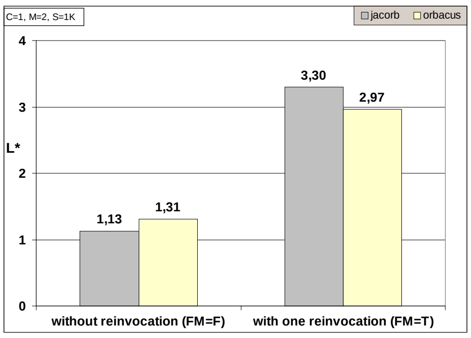
In this replication scheme, object group members are directly accessed by clients that embeds an IRL ORGW component in order to cope with member failures[20]. Figure 5(a) shows L^* values obtained by setting C=1, M=2, S=1K in both the failure-free (F_M=F) and faulty-member (F_M=T) scenarios. Note that 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. Figure 5(b) compares the stateless object group throughput in failure-free runs with the throughput measured in the basic basic benchmarks (C=1), i.e. with no fault tolerance. The throughput decrement is due to the presence of ORGW.

5.3 Stateful Replication Performance

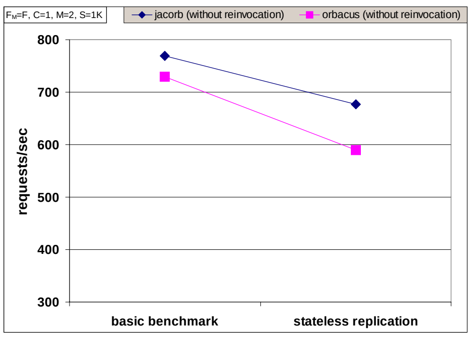
In the following experiments we measure the performances of the IRL prototype three-tier replication protocol for stateful object groups described in Section 4.3. Therefore, C clients

[19]This is the same server we replicate for testing stateful replication, then it also implements the Checkpointable and Updateable interfaces. State and updates are always empty.

[20]As ORGW is a CORBA Client Request Portable Interceptor, its performance are strictly related to Portable Interceptor Performance.



(a) Client Latency

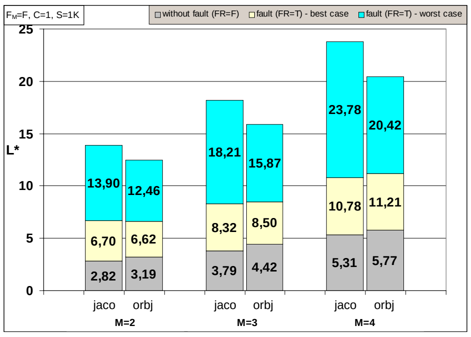


(b) Stateless Object Group Throughput

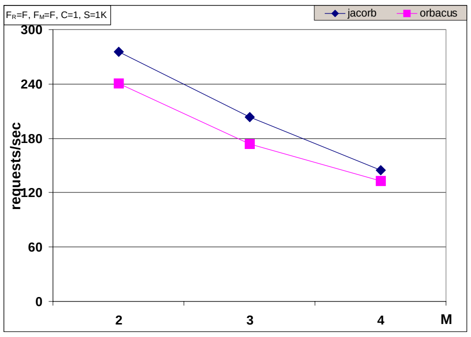
Figure 5: Stateless Replication Performance

are equipped with IRL ORGW and invoke requests of size S on the primary OGH replica that forwards them to an object group composed by M members. Each member is equipped with a co-located IRL IRGW component. In the following experiments, we vary C, M and S. We only consider primary OGH object failures (F_M=T, F_R varies) as the delay introduced by member failures is similar to the one depicted in Figure 5(a). We neither deal with host failures, as they simply introduce a delay proportional to the FN host heartbeating timeout (Figure 4(a)). Moreover, we let negligible the delay introduced by LFD and FN in order to detect OGH (object) failures.

Experiment 1. In this experiment we evaluated client latency and the throughput of a stateful object group as a function of the object group membership size in both failure-free and non failure-free runs. Therefore we set F_M=F, C=1, S=1K and vary F_R from true to false and M in $\{2, 3, 4\}$. Figure 6 shows the experimental results. In particular, Figure 6(a) shows the L* ratio values as a function of M in the failure-free (F_R=F) and mid-tier failure (F_R=T) scenarios. As the OGH recovery protocol carried out by a new primary has a best and a worst case (respectively: members are consistent after a primary OGH failure or it exists a single most updated member, see also Figure 3(b)), we draw both the minimum and maximum delays introduced by the recovery protocol. Figure 6(a) points out that, with respect to a basic benchmark (Table 1, C=1) and depending on the membership size (M), IRL takes roughly from 3 to 6 times to complete a client interaction with a stateful object group in



(a) Client Latency



(b) Stateful Object Group Throughput as a Function of M

Figure 6: Stateful Replication Performance as a Function of the Membership Size (M)

failure free runs. 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). Concerning throughput, Figure 6(b) shows the overall object group throughput of the stateful object group. It results that throughput roughly reduces of 70% on ORBacus and of 60% on Jacorb if M=2 and of 80% on both platforms if M=4 with respect to the basic benchmarks shown in Table 1 (C=1).

Figure 7 shows the percentage incidence of each IRL component in a failure-free client/server interaction with a stateful object group, as well as the absolute time values (in msec.) took by each component to perform its functionality. As predictable, the time employed by a primary

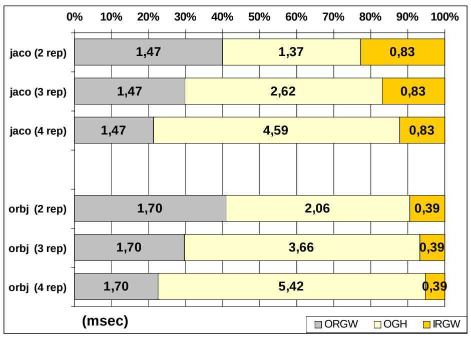
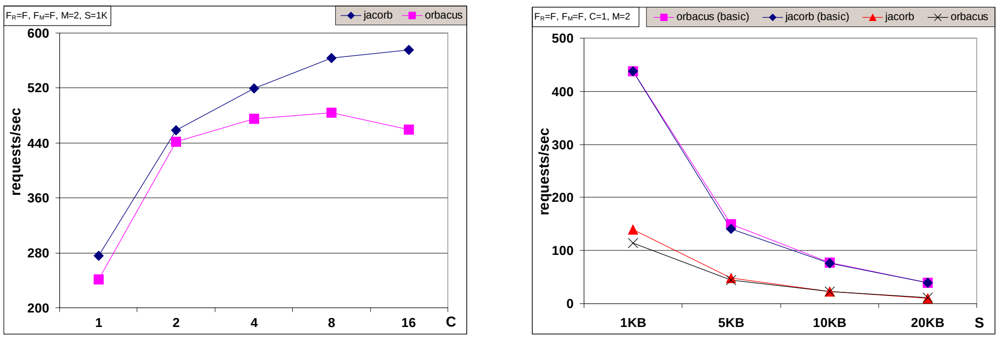


Figure 7: Percentage Incidence of IRL Components in a Client Server Interaction with a Stateful Object Group

OGH to complete an invocation linearly grows with M. Figure 7 also points out that JacORB

Portable Interceptors outperform the ORBacus ones (see ORGW times), while JacORB does not optimize co-located invocations, conversely to ORBacus (see IRGW times).

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=1K and we vary C in $\{2, 4, 8, 16\}$. The overall stateful object group throughput is depicted in Figure 8(a). By comparing these results with the basic benchmarks (Table 1), it follows (i) that throughput increases until the underlying ORB platform allows for this, and then reaches a plateau and (ii) that the throughput decreases of roughly 70% with respect to the basic benchmarks (no fault-tolerance).



(a) Stateful Object Group Throughput as a function of C

(b) Stateful Object Group Throughput as a function of S

Figure 8: Stateful Replication Performance as a Function of the Client Number (C) and Request Size (S).

Experiment 3. In this experiment we measured stateful object group 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 $\{1K, 5K, 10K, 20K\}$. The overall stateful object group throughput is depicted in Figure 8(b), in which we also plot the throughput of a simple client server interaction (no fault tolerance) while varying the client request size. As for the previous experiment, it follows (i) that throughput decreases with the request size as the underlying ORB platform does and (ii) that the throughput still decreases of roughly 70% with respect to the basic benchmarks (no fault-tolerance).

Experiments 2 and 3 shows the main implication of the IRL "above the ORB" design approach: stateful IRL object groups scale in the number of client and on the request size exactly as the underlying ORB platform does. As a consequence, performance improvements of the underlying ORB platform as well as newly available services can be easily exploited in the IRL framework.

6 Conclusions and Future Work

In this paper we have presented the three-tier approach to software replication which decouples the replication logic from both clients and servers allowing, thus, a simple deployment of server replicas on an asynchronous distributed system. Then we have presented the Interoperable Replication Logic (IRL) which is a Fault Tolerant CORBA compliant infrastructure exploiting the three-tier approach to replicate stateful CORBA objects. IRL is portable and interoperable, as it runs on several CORBA compliant ORBs and clients and server replicas lay on standard ORBs implementing only the standard IIOP protocol. We finally presented an extensive performance study of an IRL prototype, that implements a simple three-tier replication protocol based on perfect failure detection.

Concerning future work, we plan work out IRL prototype along with the following directions:
(1) using user-configurable semantic knowledge [8], in a way similar to [9], in order to enhance stateful object group performance;
(2) developing a suite of three-tier replication protocols working on partially synchronous distributed systems and on a timed asynchronous system model;
(3) introducing support for handling nondeterministic object group replicas (in a way similar to [7]) by adding a synchronization phase carried out by OGH at the end of each invocation [2].

As a consequence IRL is becoming an interesting platform for testing and evaluating software replication protocols.

References

[1] O. Babaoglu, R. Davoli, and A. Montresor, *Group Communication in Partitionable Systems: Specification and Algorithms*, IEEE Transactions on Software Engineering **27** (2001), no. 4, 308–336.

[2] R. Baldoni and C. Marchetti, *Software replication in three-tiers architectures: is it a real challenge?*, Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'2001) (Bologna, Italy), November 2001, pp. 133–139.

[3] R. Baldoni, C. Marchetti, R. Panella, and L. Verde, *Handling FT-CORBA Compliant Interoperable Object Group Reference*, in Prooceedings of the 6th IEEE International Workshop on Object Oriented Real-time Dependable Systems (WORDS'02) (Santa Barbara (CA), USA), January 2002, p. to appear.

[4] K. Birman, *The Process Group Approach to Reliable Distributed Computing*, Communications of the ACM **36** (1993), no. 12, 37–53.

[5] N. Budhiraja, F.B. Schneider, S. Toueg, and K. Marzullo, *The Primary-Backup Approach*, ch. 8, pp. 199–216, Addison Wesley, 1993.

[6] T. Chandra and S. Toueg, *Unreliable Failure Detectors for Reliable Distributed Systems*, Journal of the ACM (1996), 225–267.

[7] X. Défago, A. Schiper, and N. Sergent, *Semi-passive replication*, Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS) (West Lafayette, IN, USA), October 1998, pp. 43–50.

[8] P. Felber, B. Jai, M. Smith, and R. Rastogi, *Using Semantic Knowledge of Distributed Objects to Increase Reliability and Availability*, Proceedings of the 6th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS01) (Rome, Italy), January 2001, pp. 153–160.

[9] Pascal Felber, *Lightweight Fault Tolerance in CORBA*, Proceedings of the International Symposium on Distributed Objects and Applications (DOA'01) (Rome, Italy), September 2001, pp. 239–247.

[10] Pascal Felber and André Schiper, *Optimistic active replication*, Proceedings of 21st International Conference on Distributed Computing Systems (ICDCS'2001) (Phoenix, Arizona, USA), IEEE Computer Society, April 2001.

[11] C. Fetzer and F. Cristian, *The Timed Asynchronous Distributed System Model*, IEEE Transactions on Parallel and Distributed Systems **10** (1999), no. 6, 642–657.

[12] R. Guerraoui and S. Frølund, *Implementing E-Transactions with Asynchronous Replication*, IEEE Transactions on Parallel and Distributed Systems **12** (2001), no. 2, 133–146.

[13] R. Guerraoui and A. Schiper, *Software-Based Replication for Fault Tolerance*, IEEE Computer - Special Issue on Fault Tolerance **30** (1997), 68–74.

[14] M. Herlihy and J. Wing, *Linearizability: a Correcteness Condition for Concurrent Objects*, ACM Transactions on Programming Languages and Systems **12** (1990), no. 3, 463–492.

[15] IONA Web Site, *http://www.iona.com*.

[16] IRL Project Web Site, *http://www.dis.uniroma1.it/∼irl*.

[17] JacORB Web Site, *http://www.jacorb.org*.

[18] Dahlia Malkhi, *Quorum Systems*, The Encyclopedia of Distributed Computing (Joseph Urban and Partha Dasgupta, eds.), Kluwer Academic Publishers, 2000.

[19] L.E. Moser, P.M. Melliar-Smith, P. Narasimhan, L.A. Tewksbury, and V. Kalogeraki, *The Eternal System: an Architecture for Enterprise Applications*, Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC'99) (Mannheim, Germany), July 1999, pp. 214–222.

[20] B. Natarajan, A. Gokhale, S. Yajnik, and D.C. Schmidt, *DOORS: Towards High-Performance Fault Tolerant CORBA*, Proceedings of the 2nd Intenational Symposium on Distributed Objects and Applications (Antwerpen, Belgium), September 2000, pp. 39–48.

[21] Object Management Group (OMG), Framingham, MA, USA, *Portable Interceptor Specification*, OMG Document orbos ed., December 1999, OMG Final Adopted Specification.

[22] Object Management Group (OMG), Framingham, MA, USA, *Fault Tolerant CORBA Specification, V1.0*, OMG Document ptc/2000-12-06 ed., April 2000, OMG Final Adopted Specification.

[23] Object Management Group (OMG), Framingham, MA, USA, *The Common Object Request Broker Architecture and Specifications. Revision 2.4.2*, OMG Document formal ed., February 2001, OMG Final Adopted Specification.

[24] Fred B. Schneider, *Replication Management Using the State Machine Approach*, Distributed Systems (S. Mullender, ed.), ACM Press - Addison Wesley, 1993.