# Active Software Replication through a Three-tier Approach*

Roberto Baldoni Carlo Marchetti Alessandro Termini
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italia.
{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 replicated servers by embedding such logic in a middle-tier.*

*In this paper we first introduce the fundamental concepts underlying three-tier replication. This approach has two main practical advantages: (*i*) it allows to maintain consistency among the state of server replicas deployed within an* asynchronous *distributed system and (*ii*) it supports very thin clients. Then we present the Interoperable Replication Logic (IRL) architecture, which is a Fault Tolerant CORBA compliant infrastructure exploiting a three-tier approach to replicate stateful deterministic CORBA objects. In this context, we illustrate the three-tier replication protocol currently implemented in our IRL prototype and a performance analysis that shows the feasibility of the three-tier approach to software replication.*

**Keywords:** Software Replication, Architectures for Dependable Services, Fault Tolerant CORBA.

## 1 INTRODUCTION

Software replication is a well known technique allowing to increase the availability of a service exploiting specialized software running on COTS (Commercial-Off-The-Shelf), cheap hardware. The basic idea underlying software replication is to replicate the server of a given service on different hosts connected by a communication network so that the service's clients can connect to these different server *replicas* to increase the probability of getting replies to their requests. Software replication is commonly implemented using two-tiers (2T) as clients (client-tier) *directly* interact with a set of replicas (end-tier). If the service is *stateless* (i.e., a result to a request only depends on the current request message content), improving service availability reduces to address the issue of letting the client (more or less) transparently invoke all of the replicas before returning a "no-response" exception.

When dealing with a *stateful* service, it arises the problem of guaranteeing *consistency* among the local states of the replicas despite crashes. *Active* [30], *passive* [8], and *quorum* replication [18] are well-known 2T approaches to increase the availability of a stateful service. These approaches employ group communication primitives and services such as *total order multicast*, *view synchronous multicast*, *group membership*, etc.. However such primitives to be implementable require replicas to be deployed within a *partially synchronous* distributed system as they rely on distributed agreement protocols[1] to maintain replica consistency. A partially synchronous system is an asynchronous distributed system with some additional timing assumptions e.g., bounds on message transfer delays, process execution speeds etc. [12]. Partial synchrony can be guaranteed (most of the time) only on small size distributed systems deployed over either a LAN or a CAN (Controlled Area Network). This makes then impossible in practice the deployment of server replicas implementing a stateful service over an asynchronous distributed system such as the Internet. Let us note that replicas of a stateless service can be deployed within an asynchronous distributed system, as they do not need to run any agreement protocol.

In this paper we first introduce two desirable architectural properties for software replication, namely *Client/Server-Asynchrony* and *Client-Autonomy* that, when satisfied by a replication technique, allow (i) to deploy clients and server replicas (implementing a stateful service) within an asynchronous distributed system, and (ii) to avoid clients be involved in maintaining replica consistency. Then we propose an architecture for software replication that embeds mechanisms, agreement protocols, group primitives and data structures, needed to handle a replication scheme, (i.e., the *replication logic*) in a middle-tier *interposed* between clients (client-tier) and replicas (end-tier). As a consequence, the resulting architecture is three-tier, where the replication logic acts as a middle-tier between clients and

*This work has been partially supported by a grant from MIUR COFIN ("DAQUINCIS") and from EU IST Project "EU-PUBLIC.COM" (#IST-2001-35217)

[1] It is well known that these primitives are not implementable within an asynchronous distributed system prone to crash failures due to FLP impossibility result [17].

server replicas. Clients send requests to a middle-tier that forwards them to the end-tier, which produces a result returned to the client by the middle-tier. In such architecture, all synchrony assumptions necessary to run an agreement protocol can be confined within the middle-tier. This allows to satisfy both previous architectural properties. Other advantages of 3T replication with respect to 2T replication are investigated in Section 2.3.

In Section 3 we introduce the Interoperable Replication Logic (IRL) design, which is a CORBA infrastructure compliant with the Fault Tolerant CORBA (FT-CORBA) specification [28] that adopts a three-tier approach for the replication of stateful deterministic CORBA objects (Section 3). In Section 4 we present the three-tier replication protocol implemented as a proof of concept in the IRL prototype that has been developed in our department. Section 5 shows a comprehensive performance analysis carried out on the prototype. Finally, Section 6 concludes the paper.

# 2 SOFTWARE REPLICATION

We consider a service implemented by a set of replicas that can fail by *crashing*: a replica behaves according to its specification until it crashes i.e., it stops performing any action. Ideally, a client would like to interact with a replicated service as if it was provided by a singleton, non-replicated, highly available server. To this aim, it is firstly necessary that every replica produces the same result to the same client request. This is what the *linearizability* [20] consistency criterion formally states. Under a practical point of view, sufficient conditions to linearize the executions of a stateful replicated service are (i) *atomicity* and (ii) *ordering*. Atomicity means that if a server replica executes a state update, then each replica eventually either executes the update or fails; ordering means that each non-faulty server replicas executes updates in the same order.

## 2.1 Two-tier (2T) replication techniques

In a two-tier architecture clients directly interact with server replicas. A large number of 2T replication techniques have been proposed in the literature (e.g. [8, 30]). As examples, in the following we summarize the active and passive replication techniques as they will be used through the next sections.

In active replication, a client sends a request to a set of *deterministic* server replicas. Each replica executes independently the request and sends back the reply to the client. To get linearizability, clients and servers must interact through a *total order multicast* primitive [6]. This primitive ensures that all the server replicas process requests in the same order before failing, i.e. protocols implementing total order multicast let replicas *agree* on the order of message deliveries.

In passive replication, 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 updates the backups before returning the result to the client. 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 distributed protocol executed by the replicas to *agree* on the set of messages exchanged in a *view*[5].

It turns out that the implementation of a 2T replication technique relies on a distributed agreement protocol. Therefore, these techniques can be adopted *only if* the underlying distributed system (prone to crash failures) is *partially* synchronous (otherwise we face the FLP impossibility result [17]). In such a system, the entities involved in the agreement protocol enjoy some additional timing assumptions, e.g. bounds on message transfer delays and/or on the processes speeds. The asynchronous system model augmented with unreliable failure detectors [9] and the timed asynchronous system model [16] are examples of partially synchronous distributed systems. Implementations of total order and view synchronous multicast primitives are provided by *group communication toolkits* e.g., ISIS [7], TOTEM [26], MAESTRO/ENSEMBLE [31]. Due to the underlying assumptions these toolkits cannot be used in practice to manage large-scale groups of processes deployed over a WAN.

## 2.2 Architectural Properties for Software Replication

Apart from guaranteeing consistency, clients and server replicas implementing a replication technique should enjoy the following architectural properties to match the nature of current distributed systems (e.g. client/server architectures, n-tiers architectures):

**P1 (Client/Server-Asynchrony)** A client (resp. a server replica) runs over a distributed system without any timing assumption (i.e., an asynchronous distributed system);

**P2 (Client-Autonomy)** A client has not to participate in any coordination (distributed) protocol involving other clients and/or server replicas with the aim of maintaining server replica consistency.

P1 implies that clients and server replicas of a stateful service can be deployed across a system like the Internet as for replicated stateless service. P2 allows to implement very thin clients and widens the spectrum of devices that could host such clients (e.g. mobile phones, PDAs, etc).

Let us remark that 2T replication techniques do not satisfy P1 as both impose server replicas (and sometimes even clients) to run within a partially synchronous system. As far as P2 is concerned, for example, active replication implemented by a total order primitive where clients define the orders on message deliveries (e.g. [13]) do not satisfy P2.

## 2.3 Three-Tier (3T) Software Replication

3T software replication embeds the replication logic within a software middle-tier that is responsible for ensuring linearizability on the executions of server replicas forming the end-tier (see Figure 1). In this architecture, a client

sends the request to the middle-tier which forwards such request to server replicas (i.e., the end-tier) according to the replication logic implemented by the middle-tier. Replicas execute requests computing the results and send them to the middle-tier, which finally returns them to the clients.

To ensure the termination of a client/server interaction in 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 maintain a (replicated) state. This state could include ordering information on client requests, the middle-tier entity responsible for carrying out each request as well as its processing state etc.. Therefore, there is the need to run a distributed agreement protocol among the middle-tier entities. This implies assuming a partially synchronous model *only* for middle-tier entities. A 3T approach to replication

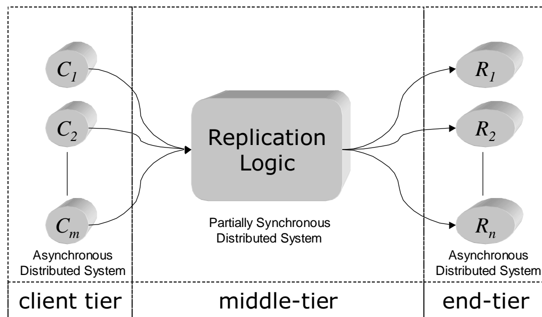



**Figure 1. Three-tier Basic Architecture**

satisfies the architectural properties defined in Section 2.3 as:

- a client interacts with the middle-tier that is the (unique) responsible for ensuring linearizability of the execution of end-tier replicas. P2 is then satisfied;

- server replicas and clients do not need to run any agreement protocol (which is run only by middle-tier entities). Clients are only augmented with a simple request retransmission mechanism to cope with middle-tier failures and arbitrary message transfer delays. For the same reasons, replicas implement simple functionality (e.g. duplicate filtering) other than providing the actual service. P1 is therefore satisfied.

Let us also remark that 3T replication produces a sharp separation of concerns between the management of the replication and the actual service implementation. This brings the following advantages with respect to 2T replication:

- a client (resp. server replica) may communicate with middle-tier entities on a point to point basis following a simple request/response asynchronous message pattern;

- the possibility of modifying the replication logic (implemented by the middle-tier) without impacting the client. For example, on-the-fly changing the replication style of end-tier replicas from active to passive [1];

- replicas are *loosely coupled*: replicas do not exchange messages among them. Furthermore, replicas can also be not aware of the replication scheme implemented by the middle-tier;

- 3T replication limits (and specializes) to the middle-tier the use of group toolkits to maintain replica consistency.

All previous points improve the scalability and the flexibility (with respect to the deployment of clients and servers) of the 3T approach when compared with 2T replication techniques. Let us also remark that a replication technique that satisfies previous properties and characteristics allows clients and server replicas to be implemented over standard technologies based on TCP such as IIOP and SOAP.

Let us finally note that the idea of interposing an indirection layer to add fault-tolerance to otherwise unreliable systems is not new. Examples include applications using gateways that accepts requests from (possibly asynchronous) clients and then forwards requests to deterministic server replicas exploiting an atomic broadcast primitive (e.g. [11, 31]). Even though the Client-Autonomy property can be satisfied by these systems, the Client/Server-Asynchrony property is not commonly met as *both* the gateways and the server replicas run within a partially synchronous system.

# 3 THE INTEROPERABLE REPLICATION LOGIC (IRL)

Interoperable Replication Logic (IRL) exploits three-tier replication to increase the availability of stateful deterministic CORBA [28] objects. IRL provides developers of fault-tolerant CORBA applications with fault monitoring and replication management functionality, which are made accessible through interfaces that comply to the FT-CORBA specification [28][2].

The decision to implement a prototype of a 3T replication system compliant with the FT-CORBA specification is due to the following reasons: (i) FT-CORBA is the best-known standard on fault-tolerance in distributed object technology and (ii) FT-CORBA enhances CORBA clients with very simple request retransmission and redirection mechanisms that actually fit the client model of a three-tier replication system. The latter point implies that FT-CORBA compliant clients satisfy the Client-Autonomy property. Furthermore, the FT-CORBA standard is the result of many research efforts carried out in the fields of distributed system fault-tolerance and of distributed object

[2]Let us point out that even though the paper shows a FT-CORBA compliant platform as a case study, the notion of 3T replication is not related to a specifi c technology.

computing. The main contributions to the standard definition have been given by the following systems: Isis+Orbix and Electra [24], OGS [14], AQuA [11], DOORS [10] and Eternal [25] (which has been the standard's proof of concept). However, such systems either do not provide replica consistency (e.g. DOORS) or to provide it they adopt the approach of relaying requests to replicas using group toolkits (e.g. Electra, AQuA, Eternal) i.e., they do not satisfy the Client/Server-Asynchrony property. The OGS system implements group communications over the IIOP message bus provided by the ORB, for which OGS assumes an asynchronous system augmented with unreliable failure detectors [9] i.e., a partially synchronous distributed system.

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

### 3.1 Overview of FT-CORBA Specification

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*[3] (FT-domain). Replicas of an object are called *members* as they are collected into an *object group*, which is a logical addressing facility ([5]) allowing clients to transparently access the object group members as if they were a singleton, non-replicated, highly-available object. If the replicated object is stateful then *strong replica consistency*[4] has to be enforced among the states of the object group members. The main FT-CORBA modifications and extensions to CORBA concerns *object group addressing*, a new *client failover semantic* and new mechanisms and architectural components devoted to *replication management*, *fault management* and *recovery management*.

**Object Group Addressing.** To identify and to address object groups, FT-CORBA introduces *Interoperable Object Group References* (IOGR*s*). An IOGR is a CORBA Interoperable Object Reference (IOR, [28]) composed by multiple *profiles*. Each profile points either (i) to an object group member (e.g., in the cases of passive and stateless replication) or (ii) to a gateway orchestrating accesses to the object group members (e.g., in the case of active replication). IOGRs are used by FT-CORBA compliant client ORBs to access, in a transparent way, object groups.

**Extensions to the CORBA Failover Semantic.** To provide applications with failure and replication transparency, FT-CORBA compliant client ORBs have to implement the *transparent client reinvocation* and *redirection* mechanisms. These mechanisms consist of: (i) upon receiving an outgoing request from a client application, a client ORB uniquely identifies the request by the REQUEST service context and uses the IOGR to send the request to a single group member, (ii) a client ORB receiving a *failover exception* (e.g. a CORBA TRANSIENT or NO_RESPONSE exception) exploits another (possibly different) IOGR profile to perform the request again until either a result is received or all of the IOGR profile had returned a failover exception (in this case the client ORB throws a NO_RESPONSE exception to the client application).

**Replication management** includes the creation, and the modification 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* (implemented by application developers) to create members, collects the members' references and returns the object group reference. *ReplicationManager* allows to select a replication technique (e.g., stateless, active or passive) for a group and to choose if consistency among stateful group members has to be maintained by the application or by the infrastructure.

**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 carried out by the *FaultDetectors*, *FaultNotifier* and by the *FaultAnalyzer* component respectively. A replicated object can be monitored for failures by a *FaultDetector* if it implements the PullMonitorable interface. *FaultNotifier* is based on the publish-and-subscribe paradigm. *FaultNotifier* receives object fault reports from the *FaultDetectors* (publishers) and propagates these fault notifications to the *ReplicationManager* and other clients (subscribers).

**Recovery Management** is based on two *mechanisms*, namely *logging* and *recovery*. *Recoverable* objects implement two IDL interfaces (Checkpointable and Updateable) to let *logging* and *recovery* mechanisms read and write their internal state. More specifically, the *logging* mechanism periodically stores on a log information (e.g., member's state/state update, received requests, generated replies etc.) of recoverable objects 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 *ReplicationManager* and one of the *FaultNotifier*. *FaultDetectors* are spread over the domain to detect failures of monitorable object group members. The overall software infrastructure is commonly referred to as a *fault tolerance infrastructure*(FT-infrastructure).

### 3.2 IRL architectural overview

IRL FT-infrastructure is the middle-tier of a three-tier architecture for software replication of (stateless and stateful) CORBA objects. Figure 2 illustrates the main IRL FT-infrastructure components, implemented as standard CORBA objects.

To preserve *infrastructure portability and interoperability* communications among clients, middle-tier and object group members (i.e., end-tier server replicas) occur through

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

[4] Formally, strong replica consistency turns out in the linearizability of the request execution histories of the replicated object.

standard CORBA invocations[5]. More specifically, clients of a stateful object group interact with the IRL OGH component that master the interactions with object group members. In the case of stateless replication, clients directly connects to a single member. To get both the termination of client/server interactions and highly available management and monitoring services, IRL components i.e., OGH, RM (ReplicationManager), and FN (FaultNotifier) are replicated. These components implement stateful services. Consistency among the state of the component replicas is maintained through common 2T replication techniques and tools. Therefore replicas of each IRL component run within a partially synchronous system and interact through an *intra-component message bus* (not necessarily the ORB) that provides group communications based on the TCP/IP protocol stack. Note that clients and object group members do not need underlying timing assumptions (i.e., IRL satisfies the Client/Server Asynchrony property). In other words, all the synchrony needed to maintain consistency is confined within the middle-tier.

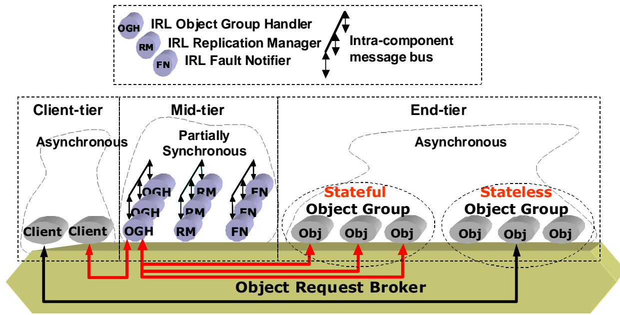



**Figure 2. IRL Architecture**

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. Interested readers can refer to [22] for further details.

**IRL Object Group Handler (OGH).** An IRL Object Group Handler 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 three-tier replication architecture. By exploiting the CORBA DII, DSI, and Interface Repository mechanisms, 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 4.1). Note that (i) clients do not belong to any kind of group and (ii) they connect to distinct OGH for accessing to distinct stateful object group. This favors the scalability of the approach.

**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 *local factories*) and returns an object group reference. Upon the creation of a stateful object group with infrastructure controlled consistency, RM also spawns a replicated IRL OGH component and the object group reference it returns actually points to OGH replicas.

**IRL Fault Notifier (FN).** As FT-CORBA compliant FaultNotifier, the IRL Fault Notifier receives (i) object fault reports from **IRL Local Failure Detectors (LFDs)**, i.e. FT-CORBA compliant FaultDetectors, and (ii) subscriptions for failure notifications from its 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 a host fault-report that pushes to RM as well as to every client that subscribed for objects running on the faulty host.

To the best of our knowledge, there are two existing FT-CORBA compliant infrastructures: the DOORS system [27] and the Eternal system [25]. DOORS does not provide infrastructure controlled consistency leaving this burden to developers. Eternal employs a group toolkit (i.e., TOTEM [26]) to maintain replica consistency and arranges *clients* and *server replicas* within the same group. This constrains both clients and server replicas to run within a partially synchronous distributed system. IRL provides infrastructure controlled consistency and satisfies the client/server-asynchrony property at the same time. The price to pay is performance. There is one additional hop to serve a client/server interaction. This point will be addressed in the Section 5.

## 4 IRL 3T REPLICATION PROTOCOL

This section illustrates the protocol implemented in the IRL prototype. The presented protocol is very simple and based on strong assumptions on the IRL Fault Notifier component (FN). It has been implemented in the IRL infrastructure as a first proof of concept of three-tier replication with the objective of taking the performance measurements described in the Section 5. We have recently designed a 3T replication protocol ([3]) based on a sequencer service (described in [4]) that does not require failure detection on OGH replicas and on object group members to enforce the liveness of client/server interactions[6].

### 4.1 Protocol Assumptions

In the current IRL prototype, OGH implements a simple three-tier replication protocol supporting static groups

[5] IRL exploits only standard CORBA mechanisms such as Static and Dynamic Skeleton Interface (SSI and DSI), Static and Dynamic Invocation Interface (SII and DII), Portable Interceptors, the Naming Service and the Interface Repository.

[6] This protocol in currently being implemented in the IRL prototype.

of both OGH replicas and object group members. Such protocol is based on the following constrains, mechanisms and properties on components and objects:

**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 and on the client application code. Additional details about ORGW can be found in [2] and [22];

**End-tier.** Each stateful object group member has to

- be deterministic and it can not invoke other objects in order to reply to a request;
- be wrapped by the Incoming Request Gateway (IRGW) IRL component that implements a simple logging and duplicate filtering mechanism to avoid multiple executions of the same client request due either to middle-tier failures or to the asynchrony of the underlying system (i.e., IRGW ensures at-most-once-request execution semantic on object group members). IRGW also implements a get_last method used by the IRL 3T replication protocol (see below). This method returns the sequence number of the last request received by IRGW (piggybacked by OGH onto the request);
- implement at least the Checkpointable and optionally the Updateable FT-CORBA interfaces.

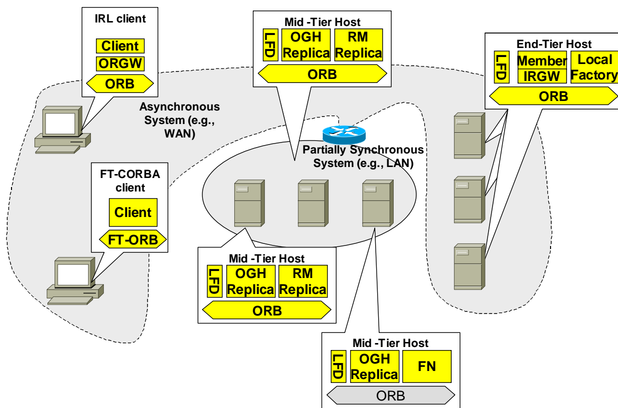



**Figure 3. Example of IRL Deployment**

**Fault Notifier (FN).** FN performs:

- *perfect failure detection* (i.e., accurate and complete in the sense of [9]) with respect to OGH replicas (i.e., FN does not make mistakes when detecting middle-tier host crashes)[7];
- *complete* failure detection wrt object group member failures i.e., FN detects all crashed members, but it can make mistakes by erroneously detecting the failure of a non-crashed object group member (because of asynchrony);

Figure 3 illustrates a possible deployment diagram of IRL.

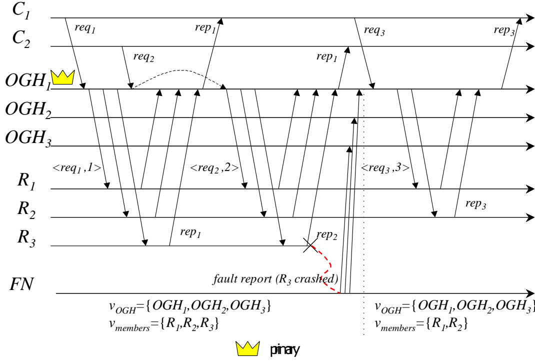
**Initialization of the 3T Replication Protocol.** When RM is requested to create a stateful object group with infrastructure controlled consistency, it provides each OGH replica with two views before returning to the client a reference containing the OGH profiles. The first view, namely $v_{OGH}$, contains the identifiers of all OGH replicas. The second view, namely $v_{members}$, contains the identifiers of the object group members. Views are updated by OGH upon receiving fault reports from FN.

**IRL 3T Replication Protocol.** OGH protocol adopts a passive replication scheme. Figure 4 shows two runs of the protocol. In Scenario 1 (Figure 4(a)) client $C_1$ issues request $req_1$ that reaches the primary ($OGH_1$). $OGH_1$ attaches a local sequence number on $req_1$ and forwards the request to every member $R_i \in v_{members}$. 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. If an object group member crashes (e.g. $R_3$ crashes in Figure 4(a)), FN detects a member failure, thanks to its completeness property, and notifies the event to each $OGH$. This allows OGH not to wait forever for a reply from a crashed member. Scenario 2 (Figure 4(b)) illustrates a run in which the primary OGH crashes. When $OGH_1$ crashes, FN notifies *each* OGH replica of the fault (thanks to the perfect failure detection property within the middle-tier). Then each OGH replica updates $v_{OGH}$ and a new primary is decided according to a deterministic rule that each $OGH$ applies on its $v_{OGH}$ (e.g. the lowest OGH identifier). In Scenario 2 $OGH_2$ is the new primary. As $OGH_1$ could have failed during the processing of a request without updating *all* the members of $v_{member}$, $OGH_2$ performs a recovery protocol, needed to ensure update atomicity, before starting to serve new client request. To achieve this, $OGH_2$ first determines if all the members are in the same state by invoking the IRGW get_last method. If all the members return the same sequence number, then they are in a consistent state and $OGH_2$ starts serving client requests. Otherwise, $OGH_2$ gets a state update from one of the members which has the greatest sequence number and sets the update to the other members[8]. Then $OGH_2$ starts

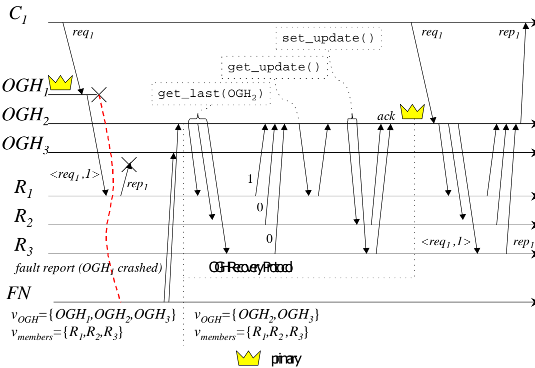
[7]Perfect failure detection can be enforced running OGH replicas either in a setting with the necessary amount of synchrony (e.g. a CAN) or in a partially synchronous system (e.g. a timed asynchronous system) with hardware watchdogs [15]. The former option is used by the current IRL prototype.

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

serving client requests. As clients implement request retransmission, $C_1$ will reinvoke $req_1$ onto $OGH_2$ as it did not get a reply from $OGH_1$. In this situation, each IRGW returns the logged result without re-executing the invocation.




(a) Scenario 1




(b) Scenario 2

**Figure 4. The IRL Prototype Three-tier Replication Protocol**

## 5 PERFORMANCE ANALYSIS

A Java IRL prototype has been implemented (it is available at [22]). In this section we show the performance analysis we carried out on such prototype. In particular, we first introduce the testbed platform and explain the set of experiments. Then we show 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 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: JacORB v1.3.21 [23] and IONA's ORBacus v.4.1 [21]. The workstations are interconnected by a 10Mbit Ethernet LAN configured as a single collision domain. As we consider

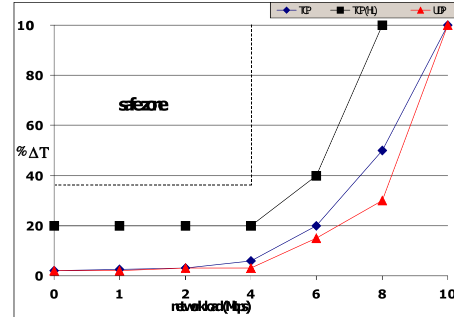



**Figure 5. FN Accuracy**

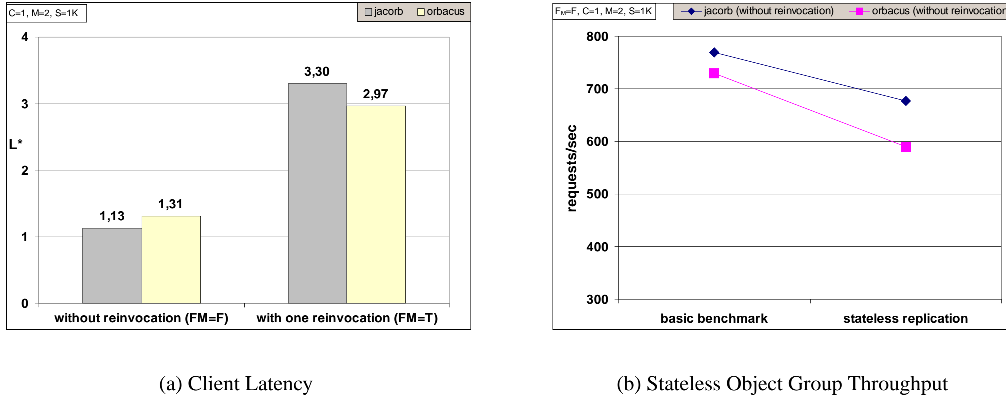
perfect host failure detection on middle-tier elements, we first evaluated the (network and processor) load conditions that guarantee this behavior. Hence, we fixed the LFD host heartbeating period and varied the network load from 0% up to 100% using a traffic generator based on UDP multicast. Heartbeats have been sent exploiting both UDP and TCP. Moreover, to evaluate sensibility to processor load, we varied the latter on the host running FN. Results are shown in Figure 5, which plots the minimum percentage increment ($\%\Delta$T) to apply to 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 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) or 1,2 sec. (if TCP is used on a highly loaded processor). All the experiments were carried out by controlling processors and network load in order that the FN timeout was always set accordingly to avoid false detections of both OGH replicas and object group members[9]. Parameters of the experiments are described in Table 1. We measure IRL average client latency and overall system throughput[10] by using a simple single-threaded server that accepts requests of variable size (S) and immediately replies.

| Params | Description | Values |
|---|---|---|
| $F_R$ | TRUE iff the test implies a middle-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 |

**Table 1. Experimental Parameter**

[9] Even if the protocol is resilient to false detections of object group members, the delay due to such events is equal to the the one due to correct detections.

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




(a) Client Latency

(b) Stateless Object Group Throughput

**Figure 6. Stateless Replication Performance**

To compare IRL latency and throughput with the standard CORBA performances, we 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 the non-replicated server instance. Results for the two CORBA platforms are shown in Table 2. 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 2.

**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 2. Basic Benchmarks**

## 5.2 Stateless replication performance

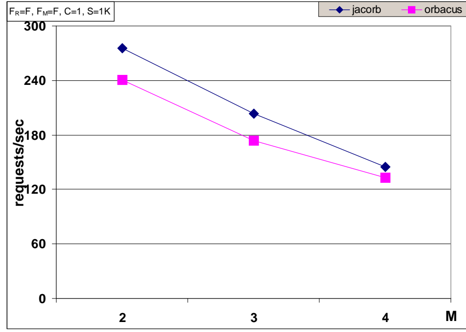
In this replication scheme, object group members are directly accessed by clients that embeds an IRL ORGW component. Figure 6(a) shows $L^*$ values obtained by setting C=1, M=2, S=1K in both the failure-free ($F_M$=F) and the faulty-member ($F_M$=T) scenarios. Note that ORGW introduces little delay in failure-free runs (about 13% on JacORB, 31% on ORBacus) and it triplicates latency upon transparently reinvoking after a member failure. Figure 6(b) plots the throughput. The decrement is due to the presence of ORGW.

## 5.3 Stateful replication performance

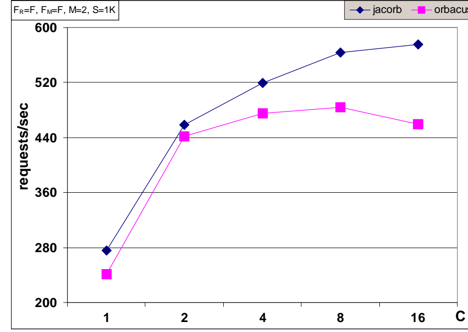
In the following experiments we measure the performances of the three-tier replication protocol described in Section 4.1. Therefore, C clients (equipped with ORGW) send requests of size S to the primary OGH replica that forwards them to an object group composed of M members. Each member is equipped with a co-located ([19]) 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 plotted in Figure 6(a). We neither deal with host failures, as they simply introduce a delay proportional to the FN host heartbeating timeout (see Figure 5). Moreover, we let negligible the delay introduced by a Local Failure Detector and FN in order to propagate the fault report of an OGH replica.

**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 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 7 shows the experimental results. In particular, Figure 7 shows the $L^*$ ratio values as a function of M in the failure-free ($F_R$=F) and middle-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 4(b)), we draw both the minimum and maximum delays introduced by the recovery protocol. Figure 7 points out that, with respect to a basic benchmark (Table 2, C=1) and depending on the membership size (M), IRL takes about from 3 to 6 times to complete a client interaction with a stateful object group in 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. Concerning throughput, Figure 8(a) shows the overall object group throughput of the stateful object group in the absence of failures. 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 2 (C=1).

**Experiment 2.** In this experiment we measured the




(a) System Throughput as a Function of M




(b) System Throughput as a Function of C

**Figure 8. Stateful Replication Performance (II)**

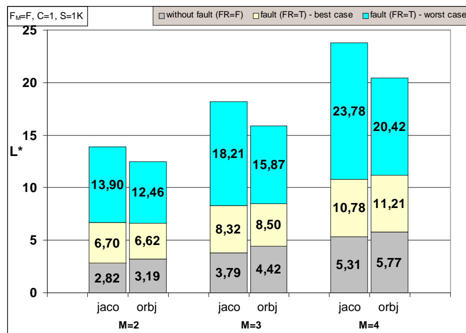



**Figure 7. Stateful Replication Performance (I): Client Latency**

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(b). By comparing these results with the basic benchmarks (Table 2), it follows that (i) throughput increases until the underlying ORB platform allows for this, then it reaches a flat and (ii) throughput reduces to about the 35% wrt the throughput measured in the basic benchmarks (Table 2).

IRL has been actually designed following an "above the ORB" approach. This brings the advantages of portability and interoperability. As a consequence, stateful IRL object groups scale in the number of clients as long as the underlying ORB platform does (see experiments 2). Furthermore, improvements of the underlying ORB platform transparently turns into IRL enhancements, as well as newly available standard services (e.g., the recent Unreliable Multicast Specification [29]) can be easily embedded in the IRL framework to achieve better performances.

## 6 CONCLUSIONS

Server replicas of a stateless service can be deployed within an asynchronous distributed system. When replicas implement a stateful service following classical 2T replication techniques, they need some additional timing assumption on the underlying system connecting the replicas to use group communication primitives. This is a real practical limit to the dissemination of replicated stateful servers on asynchronous distributed systems.

In this paper two properties have been defined, namely "Client/Server-Asynchrony" and "Client-Autonomy" that capture the possibility for a replication technique to deploy its replicas within asynchronous distributed systems and to support very thin clients. Then a three-tier approach to software replication has been presented, showing that this approach satisfies previous properties. As a consequence, clients do not interact with server replicas (each client/server interaction passes through a middle-tier entity) and the need of additional timing assumptions is confined to middle-tier entities.

We have then presented the Interoperable Replication Logic (IRL), which is a Fault Tolerant CORBA compliant infrastructure exploiting the three-tier approach to provide infrastructure controlled consistency of stateful CORBA objects. IRL is portable and interoperable, as (i) it runs on several CORBA compliant ORBs and clients and (ii) server replicas lay on standard, unmodified ORBs. Finally we presented an extensive performance study of an IRL prototype, that implements a simple three-tier replication protocol based on strong assumptions. Performance shows that 3T approach to replication is feasible.

At the time of this writing, several works are currently carried out on the IRL prototype. As examples, we are implementing a sequencer-based 3T replication protocol (described in [3]) that does not need failure detection on middle-tier entities and object group members to enforce live client server/interactions. We are also 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 want to thank Sean Baker of Iona Technologies, Benoit Foucher of ORBacus and Nicolas Noffke of JacORB development teams for the support provided during the prototype implementation. A special thank goes to Paolo Papa, Raffaella Panella, Sara Tucci Piergiovanni and Luigi Verde for their contributions to the IRL prototype.

## References

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

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

[3] R. Baldoni, C. Marchetti, and S. Tucci-Piergiovanni. Asynchronous Active Replication in Three-Tier Distributed Systems. Technical Report 05-02, Dipartimento di Informatica e Sistemistica, Università di Roma 'La Sapienza", february 2002.

[4] R. Baldoni, C. Marchetti, and S. Tucci-Piergiovanni. Fault-tolerant Sequencer: Specification and an Implementation. In P. Ezhilchelvan and A. Romanovsky, editors, *Concurrency in Dependable Computing*. Kluwer Academic Press, 2002.

[5] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[6] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[7] K. Birman and R. van Renesse. *Reliable Distributed Computing With The ISIS Toolkit*. IEEE Computer Society Press, Los Alamitos, 1993.

[8] N. Budhiraja, F. Schneider, S. Toueg, and K. Marzullo. The Primary-Backup Approach. In S. Mullender, editor, *Distributed Systems*, pages 199–216. ACM Press - Addison Wesley, 1993.

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

[10] P. Chung, Y. Huang, S. Yajnik, D. Liang, and J. Shih. DOORS - Providing fault tolerance to CORBA objects. In *poster session at IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England, 1998.

[11] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz. AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 245–253, West Lafayette, IN, USA, October 1998.

[12] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[13] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: A Fault-Tolerant Group Communication Protocol. Technical report, Computer Science Department, University of Newcastle, Newcastle upon Tyne, United Kingdom, August 1994.

[14] P. A. Felber and R. Guerraoui. The Implementation of a CORBA Group Communication Service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[15] C. Fetzer. Enforcing perfect failure detection. In *Proceedings of the 21st International Conference on Distributed Systems*, Phoenix, AZ, April 2001.

[16] C. Fetzer and F. Cristian. The Timed Asynchronous Distributed System Model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[17] M. Fischer, N. Lynch, and M. Patterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[18] D. Gifford. Weighted voting for replicated data. In *Proceedings of 7th ACM Symposium on Operating System Principles*, pages 150–162, 1979.

[19] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison Wesley Longman, 1999.

[20] M. Herlihy and J. Wing. Linearizability: a Correcteness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[21] IONA Web Site. http://www.iona.com.

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

[23] JacORB Web Site. http://www.jacorb.org.

[24] S. Landis and S. Maffeis. Building Reliable Distributed Systems with CORBA. *Theory and Practice of Object Systems*, 3(1), 1997.

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

[26] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulos, and T. P. Archambault. The Totem System. In *Proc. of the 25th Annual International Symposium on Fault-Tolerant Computing*, pages 61–66, Pasadena, CA, June 1995.

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

[28] Object Management Group (OMG), Framingham, MA, USA. *The Common Object Request Broker Architecture and Specifications. Revision 2.6*, OMG Document formal edition, December 2001. OMG Final Adopted Specification.

[29] Object Management Group (OMG), Framingham, MA, USA. *Unreliable Multicast*, OMG Document ptc/2001-11-08 edition, October 2001. OMG Draft Adopted Specification.

[30] F. B. Schneider. Replication Management Using the State Machine Approach. In S. Mullender, editor, *Distributed Systems*. ACM Press - Addison Wesley, 1993.

[31] A. Vaysburd and K. P. Birman. The Maestro Approach to Building Reliable Interoperable Distributed Applications with Multiple Execution Styles. *Theory and Practice of Object Systems*, 4(2):73–80, 1998.