Active Replication in CORBA: Standards, Protocols and Implementation Framework*

Alysson Neves Bessani[1], Joni da Silva Fraga[1],
Lau Cheuk Lung[2], and Eduardo Adílio Pelinson Alchieri[1]

[1] DAS - Departamento de Automação e Sistemas
UFSC - Universidade Federal de Santa Catarina
Campus Universitário, C.P. 476 - CEP 88040-900 - Florianópolis - SC - Brasil
{neves, fraga, alchieri}@das.ufsc.br
[2] Graduate Program in Applied Computer Science
Exact Sciences and Technology Center
Pontifical Catholic University of Paraná - Curitiba - Paraná - Brazil
lau@ppgia.pucpr.br

Abstract. This paper presents a proposal for integrating in a single CORBA middleware platform two important OMG specifications: FT-CORBA, which provides fault-tolerance support for CORBA objects, and UMIOP, an unreliable multicast protocol devised for CORBA middleware. The integration model defines a middleware support which supplies the basis to a large spectrum of group communication properties for distributed objects. Our propositions create a framework for active replication that is implemented using only OMG standards. Algorithms for reliable and atomic multicasts needed for this replication technique are presented using theoretical concepts for expressing their main features. At last, our FT-CORBA and UMIOP integration is compared to related experiences described in the literature and other active replication protocols.

1 Introduction

The FT-CORBA standard[3] [25] cames up from an OMG (Object Management Group) effort in order to specify fault tolerance for CORBA objects. The fault tolerance introduced in these specifications makes use of replication techniques. The support for managing replicated objects is provided by a set of object services. Four replication styles to distributed object are allowed in FT-CORBA specification: *Stateless* (replication technique in which the objects state are not changed by the requests); *Cold Passive* (replication with only one active object, called primary copy, which periodically checkpoints its state in a persistent storage); *Warm Passive* (similar to the previous with the difference that the primary

* This work is supported by CNPq (Brazilian National Research Council) through process 401802/2003-5.

[3] Currently at chapter 25 of the main CORBA 3.0 specification.

checkpoints its state in the backup replicas); and, at last, the *Active Replication* (where all replicas are active and perform client requests).

Actually, the current specifications provides only the abstractions for replication management. However, these specifications do not present standard solutions for executing active replicas. For example, they do not introduce interfaces or protocols for group communication, wich are necessary for active replication; they only indicate that this replication style has to be supported by a proprietary group communication tool. So, a request can be transmitted to all group members (active replicas), through an mechanism outside the ORB (Object Request Broker), using an atomic broadcast protocol [16]. If we consider the active replication technique, in a conceptual point of view, we would say that FT-CORBA provides the mechanism to manage the membership, and the proprietary tool supplies the support for group communication.

In 1999, the OMG published a RFP (Request For Proposal) defining requirements for an unreliable multicast service based on IP multicast. That RPF produced the UMIOP (Unreliable Multicast Inter-ORB Protocol) specification [24] which describes the MIOP (Multicast Inter-ORB Protocol) protocol that defines the mapping of GIOP[4] messages on the UDP/IP multicast stack. The MIOP basic function is to segment and encapsulate GIOP messages in one or more packets, and to disseminate them through the UDP/IP multicast. So, the UMIOP standard defines, inside the ORB, a mechanism of one-to-many communications with no delivery guarantees.

In order to provide group services we need a combination of protocols that deals with group management (membership, fault detection, state transfer, etc.) and with group communication (reliable, causal and atomic multicasts) offering different delivery guarantees and message ordering for distributed applications. These abstractions are being separately standardized by OMG: the group management was treated in the FT-CORBA specifications and the group communication is being developed in another OMG specification through the UMIOP. These experiences are building blocks for developing more elaborated supports providing different services (in terms of delivery guarantees and ordering of messages).

In this paper we present our experience (http://www.das.ufsc.br/grouppac) of building a support for active replication based on the CORBA object model, making use of the integration of UMIOP and FT-CORBA specifications in a single middleware platform. This resulting middleware composes a group communication support with basic primitives for unreliable multicast, where we can easily add new functionalities that provide more elaborated services of group communication. Active replication in our proposition is implemented using only OMG defined standards, in a CORBA middleware. As a consequence for supporting active replication, an optimistic atomic broadcast protocol was developed. This protocol has the following metrics: latency degree 1, message complexity $O(1)$ (in the fail-free case) and maximum resiliency ($f \leq n-1$). This protocol

[4] The *General Inter-ORB Protocol* was introduced by OMG for defining a general request/reply protocol independent of transport services.

was integrated to FT-CORBA as a Group Communication Service (GCS) using a plug-in technique. Also, any group communication tool can be integrated to a CORBA middleware using our integration framework.

The paper is organized as follows: section 2 describe a brief overview of the OMG specifications related to the concept of object group. Section 3 presents the integration model and the algorithmic support for replication that uses the OMG standards. Section 4 presents the GCS and its plug-in architecture that allows including group communication services in our FT-CORBA implementation. This section also presents the implementation description of the proposed protocols and performance measurements. Finally, section 5 reports some similar experiments found in literature and section 6 presents the paper conclusions.

2 Groups in OMG Specifications: FT-CORBA and UMIOP

The FT-CORBA specification [25] defines object services for providing the basics functionalities of fault tolerance to distributed applications. These services are divided in three main modules:

- **Replication Management (RM):** This service provides support for controlling life cycles of object groups. It offers two functionalities: *Property Management* that is used to determinate replication features (replication type, minimal number of copies, etc.); and *Group Management* which supplies mechanisms for creating group members (through objects factory) and to control groups membership;
- **Fault Management (FM):** Fault detection, notification, analysis and diagnosis are some of the attributions of this service which offers support to RM in way that this last one can keep groups membership view always updated;
- **Logging and Recovering Management (LRM):** Service used for maintaining replica's consistency; it defines mechanisms for checkpointing and replica recovering. Support for logging (used to store requests sent to the group) and mechanisms for updating members (through checkpoints) are provided by this service which is activated for recovering replica's failures through state updates.

As previously mentioned, FT-CORBA specification does not define a group communication support, which is essential in active replication. In order to provides this replication style, FT-CORBA implementations must use some proprietary tool for group communication that provides the required semantics [4, 22, 8, 17, 11].

Considering group interoperability, where object members are maybe running on different implementations of ORBs, OMG created a standardized format of references for object groups: IOGR (Interoperable Object Group Reference), which consists basically of a set of IIOP profiles identifying each group member or a proxy used for having access to the corresponding group member.

The UMIOP specification [24] introduces a protocol based on IP multicast and an object model that allow message multicasts, without reliability or ordering guarantees on message deliveries to group members. The defined object model allows a group of CORBA objects to be invoked simultaneously through a single reference, contrasting with the conventional CORBA model, where a object reference activates a single implementation.

UMIOP standards also define a group reference that addresses groups with zero or more objects. The group IOR (Interoperable Object Reference) is based on a profile type (which is different of IIOP profiles) that is adapted for sending messages using UDP/IP multicast stack. This profile contains all information (class D IP address and port number) needed for addressing group members using UDP and IP multicast protocols, and a group key for having access to member objects at ORB level. The differences between FT-CORBA IOGR and UMIOP group IOR are in their information fields: while IOGR provides the membership list, the group IOR described in the UMIOP specification contains only transport address and group key information. This group key is used in the object adapters (POA - Portable Object Adapter), available at hosts addressed by IP multicast, for getting access to group members on the host.

The FT-CORBA and UMIOP standards were implemented for us through the GroupPac [20] and MJaco [3] projects, respectively. These implementations are the basis of the work presented in this paper and can be obtained in the project site http://www.das.ufsc.br/grouppac.

3 CORBA Active Replication Protocol

Group communication with total ordering and reliability guarantees on message deliveries makes easy to maintain the consistency of group members when active replication models are used. The UMIOP standard opens an important perspective for more elaborated services of group communication that may be built using only OMG standards.

In this way, the approach presented in this paper defines all algorithmic support for active replication using OMG interfaces and standards. Our approach to build this support follows the methodology proposed in [16]: we built a reliable multicast protocol over the basic ORB communication services and an atomic multicast over that.

In the next section, we described both algorithms (atomic and reliable multicast) using the communication services provided by the FT-CORBA/UMIOP integration in our system.

3.1 System Model

Starting from the idea that FT-CORBA fills the requirements of group management and that MIOP and IIOP protocols are available in the same ORB, compounding a concrete basis for building different group communication protocols, we have to describe the adopted system model. This model defines the

assumptions and concepts used in our algorithms developed for active replication.

We consider a system model composed by a set $\Pi = \{p_0, p_1, ...\}$ of processes, where a closed group[5] $G \subset \Pi$ are defined for representing a replicated service. The considered system fills the characteristics of an *ad-hoc asynchronous system* [9] where a process fails only by stopping its activities like a crash failure (accidental or forced). In this way, our model assumes the existence of *perfect failure detectors* (class $\mathcal{P}$ [6]).

In our system we assume that processes communicate through two basic services:

- **Reliable one-to-one communication:** Defined by primitives $send(p, m)$ and $receive(m)$, this service provides a reliable point-to-point communication channels where, if a correct process p sends a message m to another non-faulty process q, then this last one will receive m;
- **Unreliable multicast:** This service is defined by primitives *U-multicast*(G, m) and $receive(m)$ where the first one is used to send a message m to members of a group. The primitive $receive(m)$ is activated by members of G for receiving a message m sent to the group. The service defined by these primitives sustains only two properties. The first one is that false messages are not created by the service. The other one gives the warranty that all correct processes belonging to a group will receive some message sent to it.

These services can model, respectively, IIOP (over TCP/IP) and MIOP (over UDP/IP multicast) protocols specified to CORBA ORBs.

Perfect failure detectors assumed in the model compose the basis for building a *group membership* service with primary-partition semantics [28]. The perfect detectors and, as a consequence, the primary-partition membership are justified in our model by the fact that the FT-CORBA specification provides these services (RM and FM). Distributed algorithms based on weaker detectors (from $\diamond\mathcal{W}$ class, for example) are too complex for being implemented in practical protocols [7], and do not reach the robustness of the ones based on perfect detectors ($f \leq n - 1$).

Our algorithms do not use the failure detectors directly, as the ones presented, for example in [6]. Instead of it, they assume that a group membership service (implemented over these detectors) will inform them of any change in the group configuration. This membership service, implemented by the Replication Manager service object in the FT-CORBA architecture satisfies the properties stated in [28], except for the view-sincrony [5].

3.2 Reliable Multicast

The reliable multicast service is defined in terms of two primitives: *R-multicast*(G, m) e *R-deliver*(m). The primitive *R-multicast*(G, m) is used to send

[5] Closed groups are those that only members can send messages to the group.

the message m to all processes of a group G, while the primitive *R-deliver*(m) is used to liberate the message to the upper layer of the correct processes of G. These primitives must satisfy the following properties [16] for characterizing a reliable multicast :

1. **Validity:** If a correct process multicasts m in G, then some correct process in G eventually delivers m;
2. **Agreement:** If a correct process in G delivers a message m, then all correct processes in G eventually deliver m;
3. **Integrity:** For any message m, every correct process in G delivers m at most once and only if m was previously multicast in G.

The Proposed Algorithm Our algorithm of reliable multicast is built using the two communication services available in our model. Also, this algorithm incorporates techniques for recovering (NACKs) and confirming (ACKs) sent messages, in a similar way to protocols like Trans [21], Psync [27] and the one used in Transis system [1].

Protocols based on NACKs are called *receiver-controlled* [19], because, in general, the receiver is the responsible for identifying lost messages and asking for recovery. Although these protocols are scalable and efficient in environments where the rate of message losses is low, they suffer from the problem of infinite buffer: the agreement property cannot be reached in asynchronous systems if the processes do not store all received messages. The stored messages are necessary for answering NACKs.

Although the use of ACKs limits the protocol scalability due to the information need about group membership, it prevents the problem of infinite buffer since sending processes can know which messages had been received by all group members and, in this way, can be removed from the *buffer* (stable messages). The use of the membership service in protocols based on ACK mechanisms is not a problem at all when we consider the FT-CORBA as part of our group support since IOGR and Replication Manager fill the characteristics of a membership control. Another problem related to protocols based on ACKS is the overhead of ACK messages : $\#G$ ACKs for each message sent to the group. This problem is minimized in our protocol by making group members to send ACKs only for each K messages received or in each T units of time. K and T are parameters of the protocol that can be adjusted in accordance with the runtime environment.

In this way, we can say that our protocolo is NACK-based, and ACKs are used only for dealing with the infinite buffer problem and to start the recuperation of message losses.

The algorithm 1 presents our reliable multicast protocol[6] where a message is sent to the group members using directly the service of an unreliable multicast (line 7). A member can accept only incoming messages not yet received (line 8). In the reception, messages are delivered to the upper layers and added to a reception buffer at the receiving member (lines 9 and 10). Lines 28 and 29

[6] The algorithm considers only one sender for better readability.

Algorithm 1 Reliable Multicast (process p)

1: {Initialization}
2: $buffer \leftarrow \emptyset$ {buffer of received messages}
3: $acks_m \leftarrow \emptyset$ {processes acknowleding message m}
4: $received \leftarrow 0$ {number of received messages}
5: $last_ack \leftarrow getTimestamp()$ {sending time of the last ACK }
6: $missing \leftarrow \emptyset$ {set of lost message ids}
Require: $R\text{-}multicast(G, m)$ {to reliable multicast a message m to a group G}
7: $U\text{-}multicast(G, m)$
Require: $receive(m)$ {receiving message m addressed to G}
8: **if** $(m \notin buffer) \wedge (m.id > maxStable(buffer))$ **then** {message not yet received}
9: $R\text{-}deliver(m)$
10: $buffer \leftarrow buffer \cup \{m\}$
11: $received \leftarrow received + 1$
12: **end if**
Require: $receive(\langle ACK, q, stable\rangle)$ {receiving ACKs }
13: **for all** $m : m \in buffer \wedge m.id \leq stable$ **do**
14: $ack_m \leftarrow ack_m \cup \{q\}$
15: **end for**
16: $buffer \leftarrow buffer \setminus \{m \mid m \in buffer \wedge G \subseteq ack_m\}$
17: $missing \leftarrow \{id \mid maxStable(buffer) < id \leq stable \wedge (\nexists m \in buffer : m.id = id)\}$
18: **while** $missing \neq \emptyset$ **do**
19: $send(q, \langle NACK, p, missing\rangle)$
20: **wait until** $(\exists m \in buffer : m.id \in missing) \vee (q \notin G)$
21: $missing \leftarrow missing_p \setminus \{id \mid \exists m \in buffer : m.id = id\}$
22: $q \leftarrow select(G)$
23: **end while**
Require: $receive(\langle NACK, q, missing_q\rangle)$ {receiving NACKs}
24: $send(q, \{m \mid m \in buffer \wedge m.id \in missing_q\})$
Require: $viewChanged(V^i)$ {membership change}
25: $G \leftarrow V^i$
26: $buffer \leftarrow buffer \setminus \{m \mid m \in buffer \wedge G \subseteq ack_m\}$
27: $R\text{-}deliver(V^i)$
Require: $(received \bmod K = 0) \vee (last_ack - getTimestamp() \geq T)$ {sending ACKs}
28: $U\text{-}multicast(G, \langle ACK, p, maxStable(buffer)\rangle)$
29: $last_ack \leftarrow getTimestamp()$

show steps for sending ACKs. Basically, an ACK contains the id of the latest stable message in the buffer of the ACK sender, i.e. the message with the highest sequence number than all its precedent messages are also received and included into this buffer. The function $maxStable(buffer)$ is used for getting the id of the last stable message. When a member receives an ACK, it adds the sender identifier (q) to all ACK lists for messages m that have $m.id \leq stable$ (lines 13-15) and removes from the buffer all received messages that were acknowledged by all group members (line 16).

During ACK receptions, lost messages are detected and requested; in line 17 the id of all lost messages are collected into the set $missing$. The while block

(lines 18-23) is where the NACKs are sent until all missing messages are received ($missing = \emptyset$). NACKs are always sent using the services of reliable one-to-one communications. Initially, the NACK is sent to the last ACK sender (in the first loop iteration - line 19). If it fails in recovering lost messages, another member is selected by a random function *select*(G) for receiving the NACK in another one-to-one communication (line 22). When the process p send a request for missing messages to some other member q, it waits until the reception of one of these messages or q fails (line 20). Line 24 presents the processing of a received NACK: the group member sends all requested messages that it has in its buffer. In order to avoid problems of infinite buffer caused by failed processes (ACKs from failed processes never arrive and the corresponding messages stay in the reception buffers forever), the algorithm makes use of the membership service that sends updated view of the group membership when it changes. A new view is received by a process when the replication manager calls $viewChanged(V^i)$. When a new view V^i is installed, all messages waiting for ACKs are checked in accordance with the new membership to verify failed members (line 26).

Proof of Correctness (sketch) The algorithm is proved correct if it satisfies the three properties of a reliable multicast:
Integrity: The non-duplication clause is guaranteed by the fact that the algorithm only delivers messages not yet received (line 8). The non-generation of messages is maintained as a direct consequence of the properties of the unreliable multicast service described in section 3.1.
Validity: This property is guaranteed by the characteristics of closed groups and by the assumption that, if a process multicast a message to your group, at least it will receive the messsage.
Agreement: If a correct process p executes *R-deliver*(m) then $m \in buffer_p$ (process p's buffer of received messages). By construction of the algorithm, m will not be removed from this buffer until p doesn't receive ACKs from all correct members of the group. Suppose that a correct process q doesn't receive m. Since q is correct, it eventually will receive an ACK with *stable* $\geq m.id$. Then q will conclude that it lost message m and then it will send a NACK requesting the retransmission of m via reliable one-to-one communication. The NACK is sent to some process that has showed evidences of having got m. Process q will receive m as long as correct processes do not remove this message from their buffers until they receive ACKs to m from all correct processes of G (q's ACK was not yet sent). This proof can be extended to all members of G, showing that all correct processes eventually receives (and delivers) all messages sent by any correct process of G.

3.3 Atomic Multicast

An atomic multicast is a group communication service which guarantees that all correct processes of a destination group deliver the same messages in the same order. This service form the basis to the technique of active replication

since it satisfies the mandatory properties for maintaining replica consistency in this replication model [30]. Formally, the atomic multicast is defined in terms of the primitives *A-multicast*(G, m) and *A-deliver*(m). These primitives are used to send and deliver messages in the group G, founded in the same properties of the reliable multicast, plus one more [16]:

1. **Local Total Order:** If any correct processes p and q of group G, deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

The group communication based on these properties are called "local atomic multicast" [16]. For providing active replication in the FT-CORBA, it is enough to use this communication service, since each object implementation is member of only one group as indicated by OMG specification [25].

Algorithm The atomic multicast algorithm presented in this text is based on the *fixed-sequencer* approach [9] where messages are sent to the group using reliable multicast as underlying service. The local total order is established by one of group members - the *sequencer* - which defines the delivery order of each message sent to correct processes in G.

The algorithm 2 presents the proposed atomic multicast protocol. In the sender, the atomic multicast is implemented by activating the *R-multicast* primitive (line 6). Message m is received by a group member (using the *R-delivery* primitive) which adds m to the *unordered* buffer (line 7). In lines 8-11, if the receiver is the sequencer, it sends a message indicating m's order (defined by the *next_order* counter). Message m is effectively delivered to the upper level when a process receives the corresponding message order $\langle id, ord \rangle$ (lines 12-20). Note that late reception of message orders may delay the delivering of messages (loop *for all* in the lines 13-20).

When a view change happens, the new membership can exclude some group members (failed processes)(lines 21-31). If the sequencer is failed, a new sequencer is evaluated through the function $getSorter(G)$ (line 22). For establishing a correct context to control message orderings, each group member sends its *next_deliver* value to the new sequencer. The asynchronous behavior of the system may determine members sending different values of *next_deliver* to the new sequencer. In the algorithm, the biggest of the received values is assigned to the sequencer *next_order*. So, line 24 indicates the new sequencer waiting messages with *next_deliver* values, from all correct processes in the new view. If the new sequencer needs to update its control of message deliveries (i.e., forcing same values to *next_deliver* and *next_order* counters), it waits the delivery of all messages with $id \leq next_order - 1$ (line 26), before distributing new message orders in the group. Following its recovery, the sequencer sends to group G orders for the pending messages in the *unordered buffer*(lines 27-30).

Algorithm 2 Atomic Multicast (process p)

1: {Initialization}
2: $unordered \leftarrow \emptyset$ {buffer of unordered messages}
3: $undelivered \leftarrow \emptyset$ {set of undelivered message orders}
4: $next_deliver \leftarrow 1$ {pointer to next message to be delivered}
5: $next_order \leftarrow 1$ {next order to be sent by the sequencer}
Require: $A\text{-}multicast(G, m)$ {to atomically multicast a message m to group G}
6: $R\text{-}multicast(G, m)$
Require: $R\text{-}deliver(m)$ {receiving message m from underlying service}
7: $unordered \leftarrow unordered \cup \{m\}$
8: **if** $getSorter(G) = p$ **then** {I am the sequencer}
9: $R\text{-}multicast(G, \langle m.id, next_order\rangle)$ {sending order of m}
10: $next_order \leftarrow next_order + 1$
11: **end if**
Require: $R\text{-}deliver(\langle id, ord\rangle)$ {receiving message order}
12: $undelivered \leftarrow undelivered \cup \{\langle id, ord\rangle\}$
13: **for all** $m : m \in unordered \wedge \langle m.id, ord\rangle \in undelivered$ **do**
14: **if** $ord = next_deliver$ **then**
15: $A\text{-}deliver(m)$
16: $unordered \leftarrow unordered \setminus \{m\}$
17: $undelivered \leftarrow undelivered \setminus \{\langle m.id, next_deliver\rangle\}$
18: $next_deliver \leftarrow next_deliver + 1$
19: **end if**
20: **end for**
Require: $viewChanged(V^i)$ {receiving new membership view of G}
21: $G \leftarrow V^i$
22: $send(getSorter(G), \langle ORDER, p, next_deliver\rangle)$
23: **if** $getSorter(G) = p$ **then** {I am the sequencer}
24: **wait until** $receive(\langle ORDER, q, ord_q\rangle), \forall q \in G$ {wait $next_deliver$ from all}
25: $next_order \leftarrow max(\{ord_q \mid q \in G\})$
26: **wait until** $next_order = next_deliver$ {waiting message orders}
27: **for all** $m : m \in unordered \wedge \langle m.id, ord\rangle \notin undelivered$ **do**
28: $R\text{-}multicast(G, \langle m.id, next_order\rangle)$ {establishing orders to pending messages}
29: $next_order \leftarrow next_order + 1$;
30: **end for**
31: **end if**

Proof of Correctness (sketch) Integrity, agreement and validity properties may be proved directly from the the reliable multicast used as underling service. So, it remains to be proved the protocol property of local total order.

Local Total Order: This proof is constructed by contradiction. Suppose two correct processes p and q delivering the same message m with different orders ($ord_p^m \neq ord_q^m$). If the sequencer is correct, this situation is clearly impossible, since each message has an unique id and the sequencer, from the algorithm itself, uses a reliable multicast for sending message orders to G's members, then $ord_p^m = ord_q^m$. Now, consider the sequencer failing during a message m multicast

to G. In that case, we have two possible situations that are guaranteed by the *agreement property* of the underlying reliable multicast:

- if some correct process received this order, all correct processes eventually receive the message order. So, the delivery order of m is the same in all processes ($ord_p^m = ord_q^m$).
- if none of correct processes received m's order sent by the sequencer, then the new sequencer will define m's order (lines 27-30).

3.4 Considerations and Optimizations for the Algorithm

Many characteristics of the presented algorithms was determinate by concepts and mechanisms defined in the OMG specifications. Even so, some optimizations may be implemented for improving the performance of the protocols. For example, we may send ACKs by *piggybacking* on application messages. This technique, largely used by other protocols [21, 1], decreases the overhead caused by control messages, increasing protocol performance in terms of application message per control messages.

In the atomic multicast algorithm, the control messages overhead can be decreased when the sequencer is also an application sender. The sender/sequencer in this case send both the application message and its defined order. Doing that, members of the group can deliver application messages as soon as they receive them.

Assuming these considerations above, the proposed protocol of atomic multicast presents message complexity of $O(1)$ (one unreliable multicast per message delivery[7]) in cases where there are no faults and message losses. The latency degree - a metric based on logical clocks [29] - of the protocol is also 1 in the fault-free case.

4 Integration of Group Communication Systems in FT-CORBA

The active replication model, proposed in the FT-CORBA specification, emphasizes the idea of a closed group of replicas accessed by lightweight clients through usual CORBA communication mechanisms (remote method invocation over IIOP). So, in this way, we proposed and developed the GCS (Group Communication Support) framework which is part integrant of our FT-CORBA implementation: the GroupPac. The GCS framework allows the FT-CORBA infrastructure to interact with the group communication system in a way transparent and standard. That is possible by having the integration of any group communication support to the FT-CORBA architecture through *plug-ins*. The figure 1 shows our integration model.

[7] Note that the ACKs are used only for bounding the size of the reception buffers and their reception does not cause message delivery in the fault-free case.

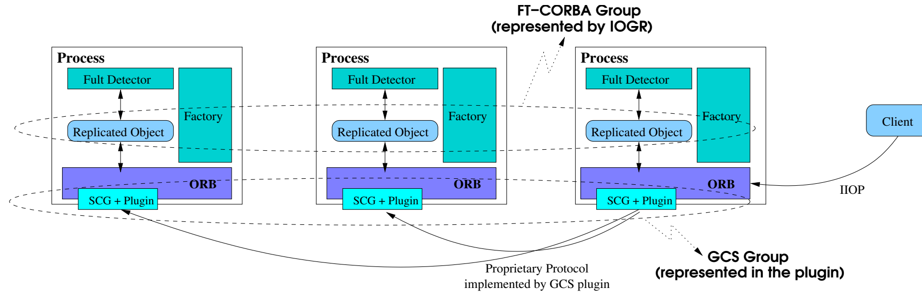

Fig. 1. Active replication in FT-CORBA.

In this figure, the client having an IOGR of a server group sends a point-to-point request (using IIOP) to one of the servers, selected by the $getSorter(G)$ function. The selected replica is used as a *bridge* for accessing protocols of group communication. Client requests are intercepted in the selected server and repassed to the GCS that multicasts this requests by making use of the group communication tools encapsulated in the loaded plug-in. Each group member receiving a request through GCS activates the corresponding method for executing the demanded service. Replies are sent back also using GCS/plug-ins and the bridge server forwards the result to the client in a IIOP one-to-one communication. So, bridge segments and plug-ins makes accessible closed group to IIOP clients in our approach for integrating group communication and active replication in the FT-CORBA.

The GCS was developed making use of instances of some *design patterns* [15] in three basic components:

- **Request Proxy:** component that receives all requests sent to the group and repasses them to the plug-in support. The proxy implementation is based on concepts of generic servers and makes use of mechanisms provided by CORBA's DSI (Dynamic Skeleton Interface) [25];
- **Request Invocator:** component that forwards the received requests from the plug-in to a local object replica of the target group;
- **Plug-in Support**: This component includes all interfaces and classes for supporting different group communication tools. Basically, we have some facilities for supporting dynamic load of *plug-ins* and some additional interfaces to group communication.

Figure 2 shows a UML class diagram with different classes and interfaces that compose the framework. In the figure, three interfaces must be implemented for the GCS plug-in: `AdaptorsFactory`, `SenderAdaptor` and `ReceiverAdaptor`.

We may include any group communication tool in the CGS framework. For that, we make use of changes in some specific fields of IOGRs. Member references are replaced by proxies references in that IOGR fields[8]. Figure 3 presents the

[8] The specifications allow this type of artifice.

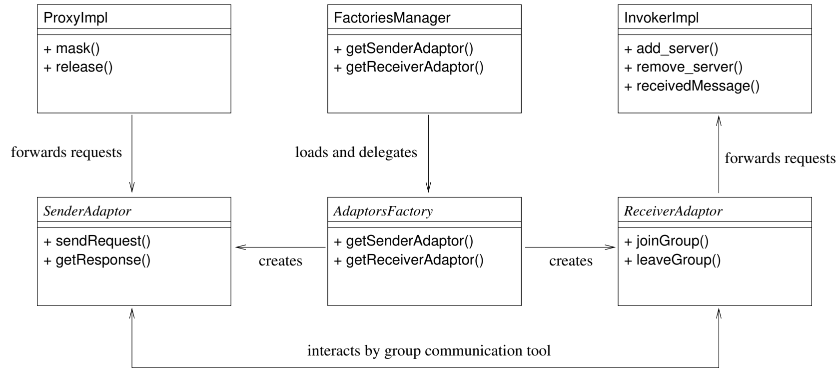

Fig. 2. CGS framework.

UML sequence diagram that shows the procedures for creating a group to active replication.

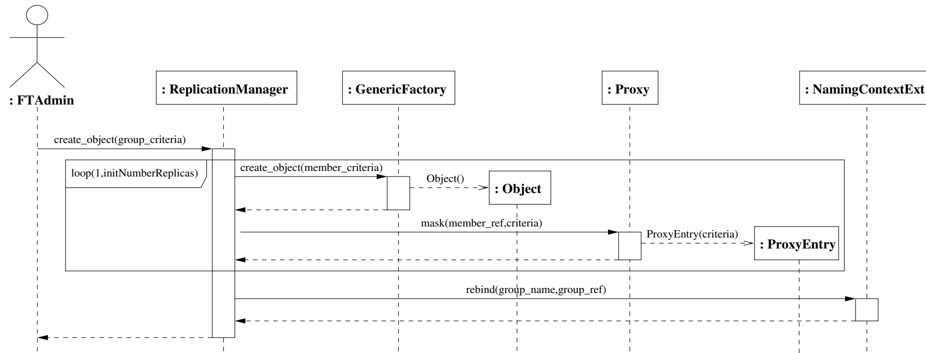

Fig. 3. Sequence diagram for an active group creation.

In the figure, we have an administrator creating a group where a parameter is passed defining some characteristics to the Replication Manager (class `ReplicationManager`). Then, this Replication Manager, for each replica to be created, makes a call to the respective `GenericFactory` that creates a replica (observing its criteria) on its location, and a call to the corresponding `Proxy`, in which it creates an entry (class `ProxyEntry`)[9]. That entry masks the associated replica. In this point, Replication Manager builds the IOGR (containing refer-

[9] For simplicity, the diagram shows only one `GenericFactory` and a `Proxy` that are activated once in each replica creation. Actually, these calls are made in different factories and proxies, which execute on different locations.

ences to the proxies) and registers it on a fault-tolerant name service [20]. The procedure stops when the IOGR is returned to the administrator.

Once the client gets the IOGR in the name service, it can invoke methods on replicas through the proxies. Such as showed in figure 1, this invocation is made in two communication steps: client-proxy via IIOP and proxy-replicas via group communication. The diagram of figure 4 presents that procedure.

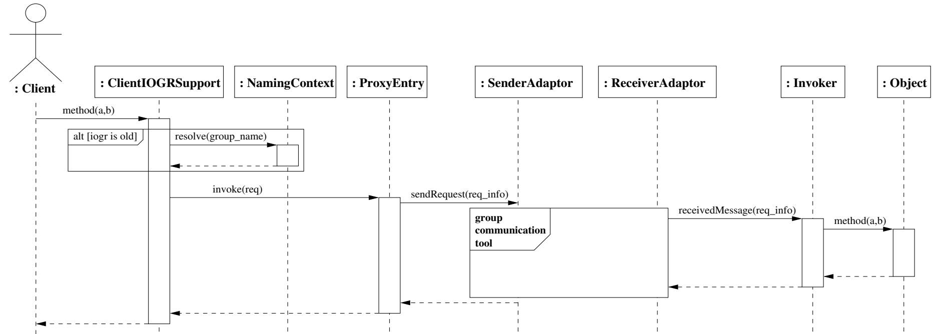

Fig. 4. Sequence diagram for an operation execution.

In figure 4, a client invokes an operation `method(a,b)` in a group that implements the IDL where it is defined. The first step to be executed in the portable interceptor `ClientIOGRSupport` installed on the client is the verification if the IOGR used is the current version (updated IOGR). Changes in the group membership need to be reflected on the group IOGR. The client interceptor makes the updating through the name service. After that, one of proxy references in the IOGR is chosen, and the request is sent to the corresponding `ProxyEntry` object (which masks the associated replica). That Proxy will be the bridge for this client. The `ProxyEntry` is a dynamic server that can handle any kind of method (via DSI). When receiving a request, it verifies which group communication support should be used for the IOGR, activating the correct client adapter (class that implements `SenderAdaptor`) to invoke that group communication tool. So, the request can be sent using a atomic multicast protocol implemented by the appropriate communication tool. That multicast will be received (in the same order) by all adapters (`ReceiverAdaptor` interface) of each replica. Adapters deliver the request message to the `Invoker`, which invokes the method on its replica. The reply of the request (if any) follow the inverse path, as showed in the figure.

4.1 MJaco Plug-in: Integrating FT-CORBA and UMIOP Specifications

Protocols presented in this paper were implemented as a GCS plug-in in GroupPac. This implementation was based on our experience with the ORB MJaco

[3]. These protocols explore the object model defined by UMIOP specification and the group management services of FT-CORBA, what implies on group communication and active replication services, all based on OMG standards. The basic motivation for building GCS like a plug-in was to have a flexible mechanism for reflecting the FT-CORBA membership of an object group, represented by an IOGR, in UMIOP group, used in the GCS context, to then, make use of the unreliable multicast services (MIOP) and reliable point-to-point communication (IIOP). The figure 5 presents the plug-in architecture.

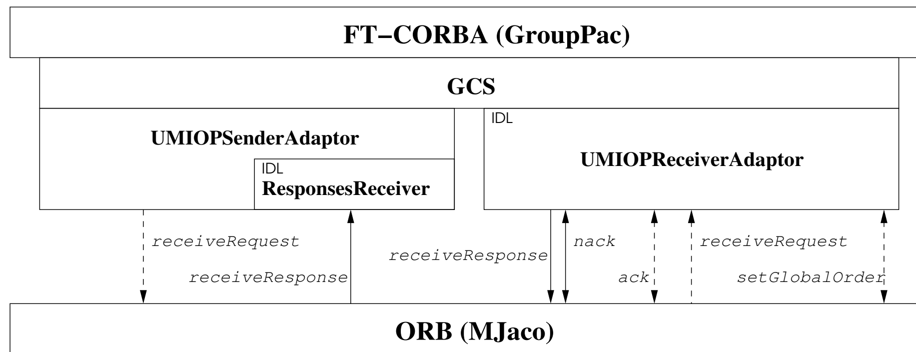

Fig. 5. Plug-in architecture.

In this figure, the plug-in components and their IDL interfaces are presented. The arrows presents the method calls and communication pattern between the sender and the receiver adaptors of the plug-in. The dashed arrows correspond to the UMIOP methods. The implementation is made from two defined IDL interfaces IDL: `UMIOPReceiverAdaptor` and `ResponsesReceiver`. The first one is implemented by all plug-in instances that are registered as UMIOP group members that are associated to FT-CORBA group replicas. This interface provides methods for message receiving as `receiveRequest()`, to reliable communications, like `ack()` and `nack()`, and for ordering, as `setGlobalOrder()`[10] algorithms. The interface `ReceiverResponses` has its implementation activated in the plug-in of the bridge replica to overcome a UMIOP objects model limitation: only oneway methods are called via MIOP. In spite of requests being multicasted in the group through the `receiveRequest()` method of the `UMIOPReceiverAdaptor` interface, replies are sent back to the bridge plug-in via IIOP using also `receiveRequest()` method call on the `ReceiveResponses` interface. The reference to the receiver object (which implements this last interface) is passed by parameter on `receiveRequest()` method, characterizing then, a callback mechanism.

[10] Note that these methods map directly to the message types used in 1 and 2.

Flexibility and the no necessity for changing internal mechanisms of the ORB are some of the advantages to implement group communication protocols as object services [12, 13].

4.2 Scalability and Performance in our Approach

In order to validate our implementation model, some experiments were developed with the GCS plug-in. We also had developed and integrated to our FT-CORBA implementation two other plug-ins to proprietary tools: Ensemble [32] and JGroups [17]. In this experiments, an example application was defined as a simple remote storage service that provides a data block storage operation which returns a *hash* of the stored data. The replication model used in the tests was based on active replicas with majority vote [11]. The protocols and plug-ins to group communication were configured in a way that they can provide atomic multicast services through our FT-CORBA implementation: GroupPac (http://www.das.ufsc.br/grouppac). The test scenario was a local network without failed object.

Two types of experiments measure the round-trip time between sending request and returning hash to/from the application service described above. Figure 6 shows the results of these experiments.

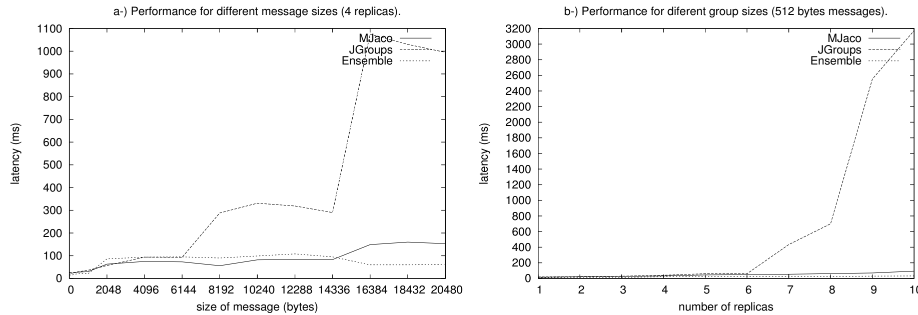

Fig. 6. Performance of active replication mechanisms.

Figure 6-a describes the response time of the replicated service considering different message sizes. In figure 6-b the response time is defined using a variable number of replicas. In both cases we have equivalent results to the plug-in using our proposed protocols and the other based on the proprietary protocols of tool Ensemble. The plug-in using JGroups communication tool get results less expressives, especially in terms of scalability.

The results presented in figure 6 are very positives, especially if we take into consideration that our implementation is a preliminary version while the

[11] The returned value to the client in that replication model corresponds to the majority value from all replica values sent.

Ensemble is a very stable tool. Another important issue is that the implemented plug-in uses only ORB communication mechanisms that, despite the guarantee of portability, impose a natural middleware overhead.

5 Related Work

A significant number of papers and works have been proposed as solutions for active replication and group communication on CORBA (see [13] for a survey), however, as far as we know, only three of these proposals are FT-CORBA compliant specifications. The Eternal system [23] uses interceptors at operational system level for accessing Totem[22], a proprietary tool of group communication. The Eternal approach ties the infrastructure to only one tool; it is not flexible and, in some ways, we may put some questions about its interoperability. The IRL project (Interoperable Replication Logic) [2] aims to implement an entirely portable FT-CORBA infrastructure that can be integrated to any ORB implementation. The IRL approach for active replication consists of a three layers architecture, considering a differentiated environment in terms of synchronism: client (an asynchronous system), intermediary layer (partially synchronous), and the replicas (asynchronous). In this context, the IRL implements replicated *gateways* in the intermediary layer where the agreement protocols can be implemented; therefore they are not subject to the FLP impossibility [14]. Those *gateways* receive client invocation via IIOP and retransmit the client requests (also via IIOP) to all group replicas. This approach allows the replicas to run on different ORB implementations, but it pays a greater cost in terms of performance, because there exists an additional step in the communication (two IIOP calls and one for executing the agreement protocol). Also, it has the problem of concentrating the *gateways* in a special network (partially synchronous network); according to authors, due to the properties of synchronism, that *gateways* mechanisms can only be implemented in local or controlled network, they can not be spread in a large-scale network. Also, the experience described in [31] utilizes replicated *gateways* to implement active replication, in a similar way to IRL and our architecture (figure 1), however it does not separate the members group from layer interactions. Also, they do not provide any consideration about the synchronism of the system, and how these *gateways* (and their active replication protocolo) are implemented.

A reliable multicast protocol that seems to be close to our algorithm 1 is Trans [21] and, in its improved version the Transis Protocol [1]. These protocols, and others such as Psync [27], use positive and negative acknowledgement (ACKs and NACKs) to assure the reliable multicast properties. They make use of piggybacking for decreasing the overhead of control messages. A point that can limit these piggybacking protocols is that they are only based on underlying services of unreliable multicast. In this case, the recoveries are subject to the rate of message losses what, in some situations, message deliveries can be delayed indefinitely for high rates of losses. Our protocol has a more deterministic

model, once it is based on reliable point-to-point communications for recovering message losses.

The literature about atomic multicast is fairly vast (see [9], an excellent survey) and then, some systems influenced our protocols. Our algorithm is based on the fixed sequencer paradigm as the protocols developed for Amoeba (method 2) and ISIS (sequencer method) systems [18, 4]. The proposed protocol in [26] explore the high probability of total order, mainly in local network, intrinsic to the multicast mechanisms used in practice. In this way, they developed an optimistic protocol which is fast when the total order is spontaneous, and relatively slow otherwise. Our protocols follow that philosophy, having a very good performance (an atomic multicast in just one unreliable multicast execution) when the sequencer is the bridge and the network without message losses. Also, we use reliable point-to-point channels for transmitting NACKs and recoveries to message losses. The algorithm developed em [10] is also based on fixed sequencer paradigm for total ordering of messages. However, it uses failure detectors $\diamond\mathcal{W}$ [6] and then, it requires at least $n/2$ correct processes for running. The main difference between that algorithm and ours is about how the system recovers a faulty sequencer: we use the membership service provided by the FT-CORBA architecture, while in [10] it is used a consensus algorithm for leader election.

6 Conclusion

This paper presents our experiences (http://www.das.ufsc.br/grouppac) for building a support to active replication using concepts and interfaces of the FT-CORBA model. The aim of our propositions was the integration of UMIOP and FT-CORBA specifications in a single CORBA middleware platform. As a consequence, for supporting active replication, an atomic multicast protocol was defined with the following characteristics: latency degree 1, message complexity $O(1)$ and maximum robustness $f \leq n-1$. The protocol implementation makes an extensive use of standardized interfaces and is based on two protocol stacks, MIOP/UDP/IP multicast and IIOP/TCP/IP, specified for CORBA ORBs. The algorithmic basis defined is implemented in the GCS framework and integrated to the FT-CORBA architecture using plug-in techniques.

A practical result of this integration is a group communication support with basic primitives for unreliable multicast, which was extended for providing guarantees for delivering and ordering (atomic multicast). Tests have shown a good performance and scalability of this support and its plug-in implementation.

The active replication in our propositions can be implemented using only OMG standards to communication and fault tolerance.

References

1. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *FTCS-22: 22nd International Symposium on Fault Tolerant Computing*, pages 76–84, Boston, Massachusetts, 1992. IEEE Computer Society Press.

2. R. Baldoni, C. Marchetti, and A. Termini. Active software replication through a three-tier approach. In *Prooceedings of the 21st Symposium on Reliable Distributed Systems (SRDS'02)*, pages 109–118, Osaka, Japão, October 2002. IEEE.
3. Alysson Neves Bessani, Joni da Silva Fraga, and Lau Cheuk Lung. Implementing multicast Inter-ORB protocol. In *Proceedings of the 6th IEEE Symposium on Object-Oriented Real-Time Distributed Computing*, Hakodate - Hokkaido - Japan, May 2003.
4. K. Birman and R. Cooper. The ISIS project: Real experience with a fault tolerant programming system. In *Proceedings of the Workshop on Fault Tolerant Distributed Systems*, New York - NY - USA, 1991.
5. Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138. ACM Press, 1987.
6. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2), March 1996. A preliminar version appeared in Proceedings of the 10th ACM Symposium on Principles of Distributed Computing, pp. 325-340, August 1991.
7. C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. A realistic look at failure detectors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, Washington - D.C. - USA, June 2002.
8. Danny Dolev and Dalia Malki. The transis approach to high avaliability cluster communication. *Communications of the ACM*, 39(4):64–70, April 1996.
9. X. Défago, A. Schiper, and P. Urban. Totally ordered broadcast and multicast algorithms: a comprehensive survey. Technical Report TR DSC/2000/036, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2000.
10. Paul Ezhilchelvan, Doug Palmer, and Michel Raynal. An optimal atomic broadcast protocol and an implementation framework. In *Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*, Guadalajara, Mexico, January 2003.
11. Paul D. Ezhilchelvan, Raimundo Macedo, and S. K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of 19th International Conference on Distributed Computing Systems*, June 1995.
12. Pascal Felber. *The CORBA Object Group Service - A Service Approach to Object Groups in CORBA*. Tese de doutorado, École Polytechnique Fédérale de Lausanne, Lausanne, Suiça, 1998.
13. Pascal Felber and Priya Narasimhan. Experiences, strategies, and challenges in building fault-tolerant CORBA systems. *IEEE Transactions on Computers*, 53(5):497–511, 2004.
14. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985. A preliminar version appeared in Proceedings of the 2nd ACM Symposium on Principles of Database Systems, pp. 1-7, March 1983.
15. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, 1995.
16. Vassos Hadzilacos and Sam Toueg. A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical report, Department of Computer Science, Cornell University, New York - USA, May 1994.
17. JGroups. JGroups: A toolkit for reliable multicast communication. Avaliable at http://www.jgroups.org, 2004.

18. M. Frans Kaashoek, Andrew S. Tanenbaum, and Kees Verstoep. Group communication in Amoeba and its applications. *Distributed Systems Engineering Journal*, 1(1):48–58, September 1993.
19. Brian Neil Levine and J. J. Garcia-Luna-Aceves. A comparison of reliable multicast protocols. *Multimedia Systems*, 6(5):334–348, 1998.
20. Lau Cheuk Lung, Joni Fraga, Jean-Marie Farines, Michael Ogg, and Aleta Ricciardi. CosNamingFT - a fault-tolerant CORBA naming service. In *Proceeding of the 18th International Symposium on Reliable Distributed Systems - SRDS'99*, Lausanne - Suice, 1999.
21. P. Melliar-Smith, L. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1), January 1990.
22. L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group comunication system. *Communications of the ACM*, 39(4):54–63, April 1996.
23. Louise E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. A. 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)*, Mannhein - Alemanha, July 1999.
24. Object Management Group. Unreliable multicast inter-orb protocol specification v1.0. OMG Standart ptc/03-01-11, October 2001.
25. Object Management Group. The common object request broker architecture: Core specification v3.0. OMG Standart formal/02-12-06, December 2002.
26. F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC'98)*, 1998.
27. Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions Computing Systems*, 7(3):217–246, 1989.
28. A. M. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. In *ACM Symposium on Principles of Distributed Computing*, pages 341–353, Montreal - Quebec - Canada, 1991.
29. André Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.
30. Fred B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
31. D. Szentiványi and S. Nadjm-Tehrani. Building and evaluating a fault-tolerant CORBA infrasctructure. In *Proceedings of the Workshop on Dependable Middleware-Based Systems (WSDM'02) - parte do DSN'02*, Washington, USA, June 2002.
32. Robbert van Renesse, Kenneth P. Birman, Alexey Vaysburd Mark Hayden, and David Karr. Building adaptative systems using ensemble. *Software - Pratice and Experience*, 28(9):963–979, August 1998.