Integrating the ROMIOP and ETF Specifications for Atomic Multicast in CORBA*

Daniel Borusch[1], Lau Cheuk Lung[1], Alysson Neves Bessani[2], Joni da Silva Fraga[2]

[1]Graduate Program in Applied Computer Science
Pontifical Catholic University of Paraná
Curitiba - PR - Brazil
{dborusch, lau}@ppgia.pucpr.br

[2]DAS – Departamento de Automação e Sistemas
UFSC – Universidade Federal de Santa Catarina
Florianópolis - SC - Brazil
{neves, fraga}@das.ufsc.br

Abstract. OMG published a draft specification for a reliable ordered multicast inter-ORB protocol to be used by distributed applications developed in CORBA (ROMIOP). This specification was made to attend the demand of applications that needed more restrictive guarantees on reliability and ordering, since there already has a specification without these resources (UMIOP). This paper presents how ROMIOP was implemented, as well as modifications that were made on the specification to make possible to implement it according to the ETF (Extensible Transport Framework) specification. Performance measures were made comparing ROMIOP with others protocols, like UMIOP, to show its characteristics and its cost.

1 Introduction

The CORBA (Common Object Request Broker Architecture) [15] architecture, standardized by OMG (Object Management Group), has the ORB (Object Request Broker) as its main component. It makes possible that objects receive and make invocations in a transparent way in distributed systems, being considered the base for the interoperability between applications on heterogeneous environments. To accomplish the exchange of messages between ORBs, there is an element that specifies a default transfer syntax besides a set of messages formats known as GIOP (General Inter-ORB Protocol). The implementation of GIOP to the TCP/IP protocol is known as IIOP (Internet Inter-ORB Protocol), which uses point-to-point communication as base, ideal for client/server applications. However, several different application areas need to disseminate the same message to an infinity of hosts. One of the ways to do

* This work is supported by CNPq (Brazilian National Research Council) and FA (Fundação Araucária) through processes 481523/2004-9, 506639/2004-5, 401802/2003-5 and FA-6651/04.

this is using multicast IP, which contains a set of extensions to the IP protocol that make possible to realize multipoint communications [4].

Since there was not a specification that described a way to use multipoint communication in CORBA architecture, in 2001 OMG published the UMIOP (Unreliable Multicast Inter-ORB Protocol) [18, 1] specification. UMIOP was proposed to provide a common mechanism to deliver requisitions by multicast, without offering deliver guarantees (reliable multicast) and even less total ordering. The standard transport protocol defined for UMIOP was multicast IP over UDP/IP, that differently than TCP/IP, is not connection guided. With UMIOP only one-way (without answer) invocations can be accomplished. Not having any kind of guarantee, UMIOP can be characterized as being of high performance, making it ideal for applications like audio and video streaming, where the loss of some packets can be tolerated. However, several applications cannot tolerate packet losses, needing more restrictive guarantees like agreement and ordering. Because of that, and since OMG had not published until that moment a specification that came with a solution, our research group proposed the ReMIOP (Reliable Multicast Inter-ORB Protocol) [3] protocol to supply that demand.

However, at the end of 2002, OMG published a draft specification, planned to be standardized on 2005. The specification introduced a solution using a multipoint communication protocol with deliver guarantee and total order (in other words, atomic multicast [5]). This specification was named ROMIOP (Reliable, Ordered, Multicast Inter-ORB Protocol) [17, 14]. ROMIOP, just like UMIOP, also uses multicast IP over UDP/IP, however invocations with return of answer (two-way) are supported. Making a detailed study of ROMIOP it is possible to verify that the specification only provides a series of IDL interfaces some of them still confused, giving a large space for interpretations. In other words, the specification does not supply details about how the interfaces must be implemented or which ordering algorithm should be used [5].

Besides this specifications, OMG recently published the ETF (Extensible Transport Framework) [16] specification, which defines a framework that allows anyone to project and implement additional transport protocol plug-ins of GIOP messages on ORB. This specification, which is already implemented in the majority of available ORBs, makes possible the extension of an ORB/CORBA with the addition of new transport protocols without having to make any significant modification in its structure. However, ETF was conceived aiming only point-to-point transport protocols. For multipoint transport, extensions in the specification are necessary.

This paper proposes, as its main contribution, a set of extensions to effectively integrate the ROMIOP and ETF specifications, including architectural and conceptual aspects besides some project decisions. The proposed solution is completely interoperable, without the use of proprietary interfaces, and totally in accordance with the OMG specifications. With this, we can consider it close enough to what could be a definite solution in terms of group communication with total order in CORBA. We also defined an atomic multicast algorithm to be implemented inside ROMIOP as a way to validate the proposed architecture, and finally we did some measures showing the cost of total ordering inside ORB.

This paper is organized in the following way: on section 2 some related works are shown. On section 3, the MJaco architecture and a set of extensions to the ROMIOP and ETF specifications are presented. On section 4, the used algorithm of total order is introduced. The section 5 presents some considerations regarding the implementation. Some results obtained with ROMIOP can be seen on section 6. Finally, section 7 presents the conclusions of this work.

2 Related Works

Group communication in CORBA was and still is a very interesting subject. The first works regarding this issue used proprietary tools of group communication. These works can be classified in the literature into three basic solutions: the approach on integration [10], on service [6, 7] and on interception [11, 12].

The integration approach consists in the construction or modification of an existent ORB, adding ways to make group processing. The main idea in this approach is that the group processing should be supported by a group communication underneath the ORB core. On the other hand, the approach that uses service objects is to provide the support for objects groups as a set of services on top of the ORB, and not as a part of the ORB. Finally, the interception approach forecasts that messages sent to the servers' objects must be captured and mapped into a group communication system, in a transparent way to the application.

In [2] it is proposed an implementation of atomic multicast over MJaco. This implementation uses the MIOP and IIOP protocols in the development of a state machine replication system [19] optimal in several aspects. A negative point about this work, in comparison with ROMIOP, is that it depends on the FT-CORBA infrastructure [13, 8], implemented through the GroupPac system.

This proposal has the virtue of having available the last OMG specifications related with multicast in CORBA. The use of these allowed us to achieve a complete interoperability and portability on ORB, fundamental requirements of any OMG specification.

3 MJaco Architecture

MJaco [1] is a CORBA middleware with group communication support based on the UMIOP [18] specifications, standardized by OMG. This middleware allows the multicast of messages in a non-reliable way, in accordance with the UMIOP standard, or reliable, implemented by the ReMIOP [3] and ROMIOP protocols, all three being based on the UDP/multicast IP stack. The integration model allows all protocols to be added to the ORB without changing the properties of portability and interoperability.

In figure 1, we have the ORB with two protocol stacks: one for point-to-point communication, based on IIOP, utilizing the TCP/IP services, and the other for mul-

tipoint communication, made by MIOP, ReMIOP and ROMIOP, utilizing UDP/multicast IP. The integration model presents several elements defined in the specification that composes the support for the two communication models.

The first stage of the MJaco project was the integration and implementation of UMIOP in the ORB. The next step on this project was the implementation of the ReMIOP protocol, which extends the UMIOP specifications with the property of reliable multicast, providing a "best effort" guarantee that all sent messages will be delivered by all correct processes of a group. Finally, ROMIOP was added to the set of protocols, being the only one that, besides having reliable multicast, provides total order message delivery, in other words, all correct members deliver all messages in the same order.

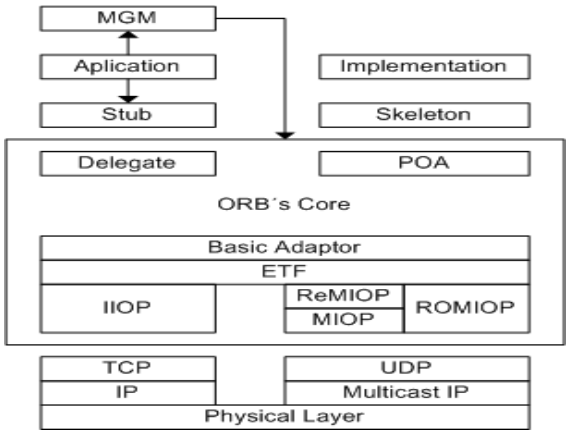

Figure 1. MJaco architecture.

It is important to note the way ROMIOP was introduced in MJaco. Instead of putting it on top of ReMIOP, or even in the same level as ReMIOP using MIOP as base, it was developed from the bottom. This, as will be shown later on, was made due to considerable differences on the format of the specifications of these protocols, being preferable to start its implementation without using almost anything from MIOP. It is also important to clarify that the ETF specification does not easily allows stacking protocols.

3.1 ETF Specification

ETF [16] is a specification of a platform that enables third parties to project and implement messages transports plug-ins. With it, several middleware that implement CORBA become much more flexible. This happens because there is in the specification all interfaces and methods that must (or may, on the optional ones) be implemented and which functionality each one must have, making unnecessary to modify the ORB code. This ensures the proprieties of portability and interoperability of the ORB.

The biggest problem of this specification is that it only defines how to add point-to-point transport plug-ins, which makes it deficient to multipoint protocols, like ROMIOP. Due to this fact, the chosen middleware had to be slightly modified, adding the functionality of sending messages to groups.

Basically, the specification defines four mandatory interfaces: connection, profile, listener and factories. The first one splits the message layer (GIOP) of the ETF layer, creating an interaction channel between messages and connections (both from clients and servers). The second one stores all the information related to the protocol, including methods to send (marshalling) and receive information through the IOR. The third one provides the initiative to "be connected" to a requisition made by a client, directing this requisition to a server. The forth and last interface is responsible to create instances of clients, making the connection of the ORB with the plug-in (the figure 5, on section 5, presents the ROMIOP implementation as an ETF plug-in).

A sequence of steps showing the interaction of a plug-in and an ORB can be seen in the figure 2 bellow:

1.	{Creating Server Listener}		
2.	ORB	->	Factories.create_listener()
3.	ORB	->	Listener.set_handle()
4.	ORB	->	Listener.listen()
5.	ORB	->	Profile.marshal()
6.	{Creating Client Connection}		
7.	ORB	->	Factories.demarshal()
8.	ORB	->	Profile.is_match()
9.	ORB	->	Factories.create_connection()
10.	ORB	->	Connection.connect()
11.	{Creating Server Connection}		
12.	Handle.add_input()	<-	Plugin
13.	ORB	->	Listener.accept()
14.	{Request Messages Receipt at Server}		
15.	Handle.data_available()	<-	Plugin
16.	ORB	->	Connection.read()

Figure 2. Interaction steps between the ORB with the plug-in.

At first there must be an instance of the server running that stays waiting the creation of client connections (line 1). All these steps are made by the ORB, starting with the invocation of the method `create_listener` provided by the `factories` interface (line 2). It creates an object that implements the `listener` interface and it is returned to the ORB. In the next two lines (3 and 4) the ORB uses the return of line 2 to invoke the methods `set_handle` and `listen`. The first one simply allows the plug-in to callback the ORB whenever it is necessary while the last effectively allows this instance to receive requests. Lines 5 and 16 indicate that at any time the profile of this transport can be marshaled (be serialized).

The next stage to make a communication is the creation of a connection requested by the client (line 6). All these steps are made by the ORB, starting with the invocation of the method `unmarshal` provided by the object that implements the `factories` interface (line 7). This happens just after the protocol being used is identified by the information contained in the received IOR. This invocation returns an object that implements the `profile` interface, that is used by the next line of code (line 8) to verify if it does not already exists an equal profile created before, meaning that the connection has already been opened before. If the connection does not exist, then both following instructions (lines 9 and 10) will be called. The first one, provided by `factories` interface, creates a connection, while the second uses the return of the first one to effectively enable the connection.

The third stage is the creation of the server side of the connection, which always happens when a client requests a new connection (line 11). There are two distinct possibilities to accomplish this function: by "callback" and by "polling". The first option (line 12) is initiated by the plug-in that calls the method `add_input` of the `handle` interface that was received as parameter on line 3. On the other hand, the second option (line 13) is initiated by the ORB invoking the method `accept`, which stops the thread until there is a connection.

The last stage shows the receipt of a request message by the server side (line 14). After the establishment of the connection in the last stage, the `listener` instance can signal to the ORB that there are new data available, calling the method `data_available` of the `handle` interface (line 15). Next, the ORB can read these data through a `read` method (line 16) of the `connection` interface.

3.2 ROMIOP Specification

ROMIOP (actually a draft [17, 14]) defines a set of interfaces to provide a multipoint communication with deliver guarantee and total order for every non-faulty members of a group of objects. It supports both requests that need replies (two-way requests/replies) and the ones that do not need (one-way requests). This specification was projected so that the protocols that implement it could coexist with IIOP, MIOP and any other multicast communication protocol, not being able to interfere in the functioning of them.

The specification defines an interface that configures the several available methods to consolidate replies and the quality of the ordering service. Regarding the first factor, there are two distinct ways to accomplish the consolidation: **simple voting**, where there are three possibilities to determine the reply (first received, last received and the first that satisfy the parameter of data consistency); **quorum voting**, where there are two possibilities to determine the reply (number of members that send the same reply and percentage of members that send the same reply), both dependent of the consistency parameter.

The data consistency parameter settle three possibilities: all the replies must be the same; all the replies must be different (apparently without any utility); and the standard, in which the majority of the same replies prevail. It is interesting to note

that with this last option it is possible to provide a limited support to fault tolerance, using the idea of state machine replication [19]. Beside that, there is an additional parameter related to the reply consolidation that ends up overcoming all the others. It is possible to configure a timeout to the receipt of the replies. With this enabled, even if the chosen option has not yet being accomplished, the consolidation process is forced with the already obtained results.

The specification also defines how the consolidation and notification of replies must work: to each request message sent to the objects group there must be created an instance of a class responsible for the receipt of its replies. This instance is responsible for determining if the reply is a success, if a timeout happened, if there is no sufficient quorum, if the voting was inconsistent or if a key member was missing.

Another foreseen interface in the specification is the group service that provides basic operations like add and removal of members, besides the creation of groups.

To finish, speaking about the communication protocol, ROMIOP defines only formats for several types of messages. These formats consist in a header that has a fixed size and brings information related with the data contained in the packet (like it is type, number of identification, position of the packet inside a set of messages, etc.) and an area of data. The defined types of packets are `Request`, `Reply`, `ACK`, `NAK`, `KeepAlive`, `Cancel` and `MemberChange`. Each type has a different structure in the data area, following the semantic of the packet. Between several fields we can quote: to which member the packet is directed, address of ACK/NAK, ID of the already received packet, status of a member, etc.

Each one of the packets described above always come after a standard packet header (structure `PacketHeader`) that inform, besides the type of the following packet, complementary information, like if there is the need to send an ACK after the receipt of the same, the size of the packet unique identifier, version of the protocol, etc. Figure 3 shows the definition described above.

```
module ROMIOP {
     typedef   octet              PacketType;
     const     PacketType         Request = 0;
     const     PacketType         Reply = 1;
     const     PacketType         ACK = 2;
     const     PacketType         NAK = 3;
     const     PacketType         KeepAlive = 4;
     const     PacketType         Cancel = 5;
     const     PacketType         MemberChange = 6;
     const     PacketType         MsgOrder = 7;
     struct PacketHeader_1_0 {
               char               magic[4];
               octet              version;
               octet              flags;
               short              packet_type;
               unsigned long      packet_size;
};  };
```

Figure 3. ROMIOP packet definition.

3.3 Proposed Extensions for Integrating ROMIOP and ETF

Since, until the moment, the ROMIOP specification is still not finished, several considerations had to be taken and added so that the implementation of this specification became achievable. The most important one was the creation of a new type of message, the one who carries the messages orders sent by the leader server (`MsgOrder`). Its definition is very simple: there is an identification of who sent the message and a list with a structure, which was also created, that contains the identification of the message and the identification of the member who sent the message.

The message order to be delivered is exactly the same of the list. It is necessary to send both the message identification and the member who sent it, because in the cases were there are messages with the same identification, the member who has the lesser identification will have its message delivered first.

Another crucial point is related to the reply consolidation. It is only said which models must be implemented (simple or quorum voting), however it is not informed where the consolidation must occur (in the sender or receivers). In this implementation, the consolidation algorithm is processed in the client who has made the request, taking out of the servers this extra load of processing. This issue could be considered the most laborious to be developed.

Since the fact that there is not any module related with the reply consolidation in the ETF specification, as well as a lack of related work, the entire model had to be developed to solve the problem. Several approaches where analyzed, implemented and tested before reaching the definitive method that attended the incomplete ROMIOP specification. Because of this fact that the protocol did not use as base any other existent protocol (see figure 1 at section 3). The control of the received replies is not of the protocol. The middleware is the only one who defines which process (client) owns the reply. This approach had to be detoured, since with this, only the first reply would always be used, getting rid of all the others (this fact would always occur in the existence of more than one server). The chosen solution was to created a "deviation" of this information to the ROMIOP protocol, after the one being sent to the middleware, attending this way both specifications (ETF and ROMIOP).

Although it may look a slow method (the same information passes through the protocol twice) the tests that were made (see section 6) prove that the performance loss was insignificant. Beside that, this method has the advantage of modularizing the functions: messages that have to be consolidated go through this process, while the ones that do not need, follow a direct path. With this, each client process the consolidation of its requests while the servers stay with the task of replying the requests and defining the order of the messages to be delivered.

Exactly because of the need to consolidate the replies that were implemented a member service (membership) with more functionality than the one already existent in MJaco. The protocol needed to know exactly the quantity of functional members so that the client knows how many replies it must wait for. This entire module was projected without any kind of specification. Since the protocol needs to keep an updated list with the members, and it must also be capable of handling omission faults, crash faults and problems with the physical network (like a cut network cable or a

badly configured router), it was implemented an algorithm that is executed by the leader server. With it, at each pre-determined time interval, a message is sent to every member of the group asking if they are still alive. All the members that do not send an ACK to this message will be removed from the group, in other words, it is assumed the perfect detector abstraction.

Finally, the specification in its current state does not define the total ordering algorithm to be used. With this, it was defined an algorithm based on the fixed sequencer [5], presented in section 4, so that it became possible to verify the potentialities of the proposed architecture.

4 Atomic Multicast Algorithm

A total order algorithm with reliable multicast is the one that guarantee that all non-faulty members of a group will deliver the same set of messages in the same order [5]. This type of algorithm is also called of atomic multicast, because the deliver of a message happens as an indivisible primitive: the message is either delivered to everyone or to no one, and if delivered, all the other messages will be ordered either before or after this one.

This type of ordering makes easy the maintenance of a consistent global state between several processes, being used as a base to the implementation of fault tolerant through the active replication (the current state is replicated) [19].

It is important to note the way that the packet unique identifier that each member puts in every message sent to the group is used. This ID is nothing more than a local counter that is started with the number zero. Every sent packet increments this counter. Also, every packet that any member receives, even if is not directed to him, the ID is analyzed. If it is bigger than its local counter, than this value becomes its local counter number. If it is the same or less, nothing is done. With this it is possible to use the classic algorithm of events ordering of [9], which allows the identification of the order in local messages (the message with the lesser identifier will always be delivered before the message with the bigger identifier).

4.1 Assumptions and System Model

Regarding the process failures, we assume a crash fault model. The group service module tolerates processes faults since it keeps an updated list of members.

We assume reliable channel, this semantic is implemented by the periodic retransmission of messages until every receiver processes acknowledge the receipt of the message (through an ACK message). Duplicated messages are detected by its identification and are discarded, however ACKs to these messages are sent before the discard occurs, since its receiver may lose the ACK.

Assumptions regarding time had to be made to implement the ROMIOP protocol. To determine the re-send of the messages because of the non-receipt of enough ACKs was adopted that the sum of the times of computing and communication are

synchronous, in other words, there is an upper limit known so that both occur. Another presumption taken regarding time was of a perfect failure detector. If the time taken to the reply of the message asking if it is alive (`KeepAlive`) arrives after a certain value (or never arrives), that process will be considered as being with problem and will be excluded from the group.

If any process locks and do not return (crash) there will be no problem, since the membership module keeps a list of the quantity of members updated. Another possibility is the lock of a process and its return after a period of time. In this case, depending of the time it stays without answering it may be removed from the group, however it will be re-included in the group members when it returns. The only problem related to this last possibility is that the receivers will discard any reply, from the excluded member, originated from a requisition delivered before the lock happened.

Finally, it was implemented a leader election algorithm in which the first process that enters the group is considered the leader. This functionality was implemented by simply sending a message to the group and waiting for an answer by a determined period of time. If no one replies, it considers itself as the leader and starts to warn every other process that enters the group of the existence of a leader. Notice that every timeout parameter, related to these times, are configurable so that the protocol can work in the best possible way in each network environment.

4.2 Algorithm

The ROMIOP adopted protocol for atomic multicast is based on the fixed sequencer paradigm [5]. Basically, the protocol works in the following manner: the emitter casts a message requesting the members of the group to enter this group. Next, it can send message requests to the group address and it stays waiting for a quantity of ACKs equal to the quantity of receivers in the group. If the quantity of ACKs is lesser than the expected, the request is re-send to the group. The ROMIOP simplified algorithm is shown in figure 4.

Initially, all buffers are initialized with empty values (line 1 to 5). To multicast a message to the group (line 6) they need first to be stored (line 8) in a local buffer (excluding the ACK type of message). After the message is stored, a timer is created (line 9). Just after both these steps that the message is sent (line 10). This is needed because the messages have to receive confirmations (ACKs) that it effectively reached its destination. If an enough number of confirmations do not come, the message is re-send after the finish of the timer. Upon the receipt of ACKs (line 41), if they are sufficient (line 42), the timer to that message is stopped (line 43) and the buffer that stored the message is deleted (line 44).

Every received message has a different type of processing (line 15). The requests (line 16), only received by servers, are stored in a local temporary buffer (line 17). These requests are only delivered upon the reach of the message with the order of the messages (line 33). After the receipt of the order, the servers compare its identifications with the ones that they have stored in its local buffer of already received messages (line 35). All messages that it has, starting from the first one and going se-

quentially to the last, will be delivered (line 36). If, by any reason, the server does not have one of the messages contained in the order, all the subsequent messages will not be delivered until the missing one is received.

```
1.  {Initialization}
2.  buffer <- ϕ {received messages buffer}
3.  received <- ϕ {received and not delivered requests buffer}
4.  order <- ϕ {order messages of request to be delivered buffer}
5.  members <- myself.id {group members list}

6.  {To R-multicast(m)} {group message diffusion}
7.  if m.type = REQUEST | REPLY | MEMBERCHANGE | MSGORDER then {m.type:
    type of message m}
8.        buffer <- buffer U {m}
9.        timeout(m, 2000) {start timeout to re-send if needed}
10.       send(m) {send the message}
11. end if
12. else if m.type = ACK then
13.       send(m)
14. end if

15. {To R-receive(m)} {receipt of a message}
16. if m.type = REQUEST then
17.       received <- received U {m}
18  end if
19. else if m.type = REPLY then
20.       consolidate(m) {consolidate the received replies}
21. end if
22. else if m.type = MEMBERCHANGE then
23.       if m.status = added then {m.status: type of member change}
24.             members <- members U {m.id}
25.       end if
26.       else if m.status = removed then
27.             members <- members \ {m.id}
28.       end if
29.       else if m.status = online then
30.             R-multicast(members)
31.       end if
32. end if
33. else if m.type = MSGORDER then
34.       order <- order U {m}
35.       while m.id ∈ received.id do
36.             deliver(received) {deliver the message in the correct
    order}
37.             order <- order \ m
38.             received <- received \ m
39.       end while
40. end if
41. else if m.type = ACK then
42.       if enoughACK(m) = OK then {if received an enough number of
    ACKs}
43.             timeout(m).stop {stops the timeout that re-sends the
    message}
44.             buffer <- buffer \ m
45.       end if
46. end if
```

Figure 4. ROMIOP simplified algorithm.

The leader server sends the message with the order of the messages (`MsgOrder` type). Always after receiving a request message, a configurable timer is started. After

the end of this timer, it is sent to the other servers belonging to the group the order message, containing the identification of all received requests messages during that time.

Just after the processing of the request message by the server, if it needs a reply, a reply message type will be sent to the group address (line 6). The emitter client who sends the request message stays waiting for a certain number of reply messages, depending on how the consolidation was configured (line 19 and 20).

Finally, every time an object wants to enter the group, it sends a message of type `MemberChange` with the status `added`. All participants of the group that receive this kind of message (line 23) add the member who sent the message to its local list. After the member receives the confirmation that he entered the group, it sends another `MemberChange` type of message, but with the `online` status. When the members receive this message (line 29) they reply sending its local list of members.

5 Implementing ROMIOP below ETF

To better present the protocol and the way it was adopted to be implemented, in order to satisfy the requirements of the ETF specification, follows the figure 5. It is a class diagram showing only the extremely essential methods and attributes, besides representing only the most important relationships between classes.

The section 3.1 already describes the steps to create the `ClientROMIOPConnection` and `ServerROMIOPConnection` classes. Both are derived from the `ROMIOPConnection` class, being the first one created by the `ROMIOPFactories` while the second one by `ROMIOPListener`.

Basically, at the reception of any packet message, through the `ROMIOPConnection` class, the fragment, after several verifications, is added to an object of the `FragmentedMessage` class. When a message is considered complete it is then delivered to the ROMIOP algorithm, where it will be ordered.

For messages that need reply, the `ReplyConsolidatorImpl` and `NotificationImpl` classes are used, being the second controlled by the first, and this controlled by the `ROMIOPStrategy` class. The `ReplyConsolidatorImpl` effectively consolidates the reply and it is activated in the creation of the request message that needs the reply or in the receipt of any of its replies. The `NotificationImpl` class is only used to notify the ORB of the chosen reply.

Finally, the `ROMIOPStrategy` class keeps all the remaining necessary processes for the correct functioning of the protocol, being for this considered the most complex class and the one with more functionality. Between some of its functions are the send of ACKs, the membership control and the send of the message order.

Almost every protocol class use both `MulticastUtil` and `ROMIOPProfile` classes. Each one of them is responsible for storing specific information. The first one is related with the whole protocol, storing protocol configurations, like the consolidation method and the time limits. The second one is connected uniquely with a connection, being responsible for storing information of it, like the group address.

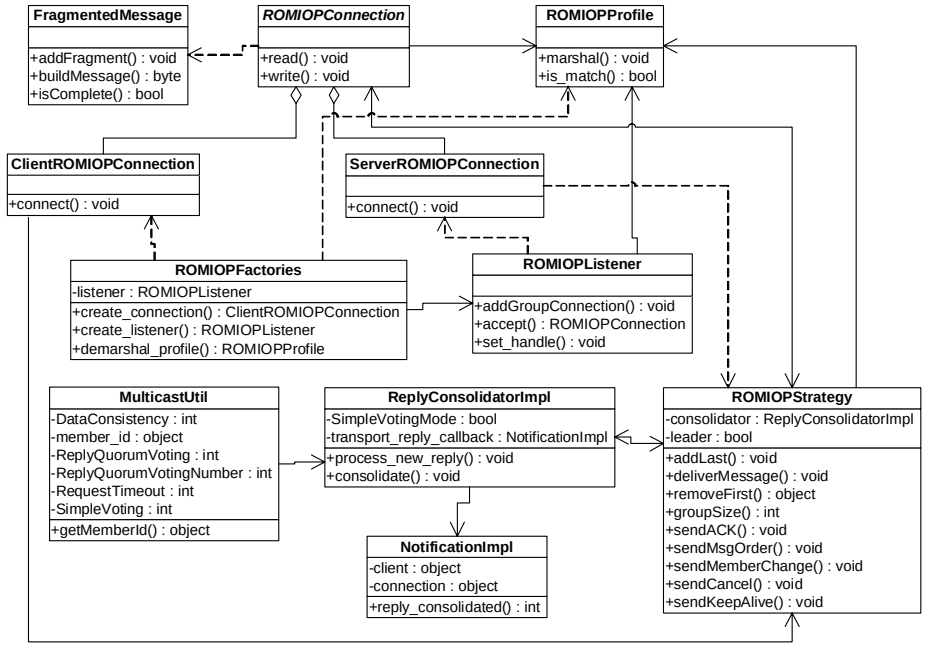

Figure 5. ROMIOP plug-in simplified class diagram.

6 Performance Evaluation

With the purpose of analyze the MJaco performance with the ROMIOP, as well as the choices made in the implementation of it, there were made several tests. The environment in which the tests were done was a set of machines running Windows XP as operational system. The client was executed in an Athlon 2600+ machine, with 512MB of RAM memory. The leader server was executed in a Pentium 4 at 2.6GHz with 1,5GB of RAM memory. The other two servers were executed in Athlon machines at 1.47GHz with 248MB RAM memory each.

The first test had the objective to analyze the scalability as well as the velocity of the algorithm. The principle is simple: a message with a variable size is sent and it stays waiting for a reply (an integer of 4 bytes). The final reply (the one that the client will actually use) is only consolidated after the receipt of all replies of each server (atomic type of consolidation). In figure 6 the result can be seen.

The result was exactly like the expected one. As much as the number of servers grown, the time that the client takes to consolidate the replies becomes bigger. Another point of interest is the low increase of cost with the insertion of more servers, proving that the adopted protocol is relatively scalable. It is important to say that the time the leader server takes to send the message order was configured to 10ms. Finally, it can be easily seen a risen in the time taken to accomplish the consolidation

from 1 to 2 or 3 servers. This happens because the message order is not send with only one server.

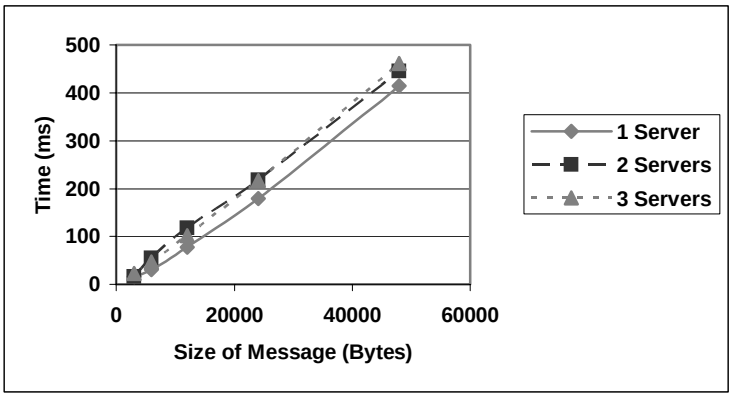

Figure 6. Test result of the atomic consolidation.

The following test was made to analyze the consolidation algorithm, where the atomic option was compared with the first reply option. The figure 7 shows the results for 2 and for 3 servers.

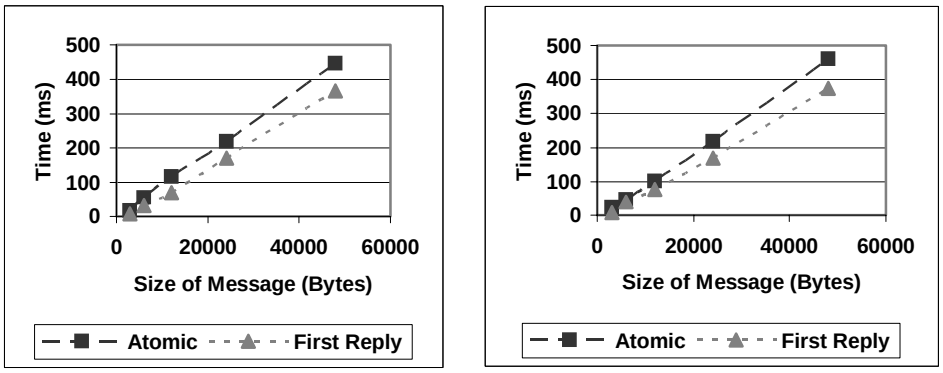

Figure 7. Comparative with two types of consolidation methods with 2 and 3 servers.

The analysis of the graphics brings, besides the expected that the time taken to consolidate only the first reply is smaller than the one with all (atomic), the fact that for small messages (approximately until 3000 bytes) there is practically no performance improvement, being for this cases considered more advantageous to use the atomic consolidation, since it brings more security. With this result, it can be seen that for large types of messages it is crucial the choice of the consolidation method. If the application needs to support Byzantine faults or if you only desire to be sure of the obtained reply (like life support systems or military applications) there will be a considerable cost. Notice that even with the increase of the number of servers (from 2 to 3), the time to obtain the reply from the first reply consolidation method practically does not increase.

Next it was made a test to analyze one more consolidation option that ROMIOP implement. It was tested the percentage of members quorum system. The chosen value configured to make the consolidation was of 51% of members (figure 8).

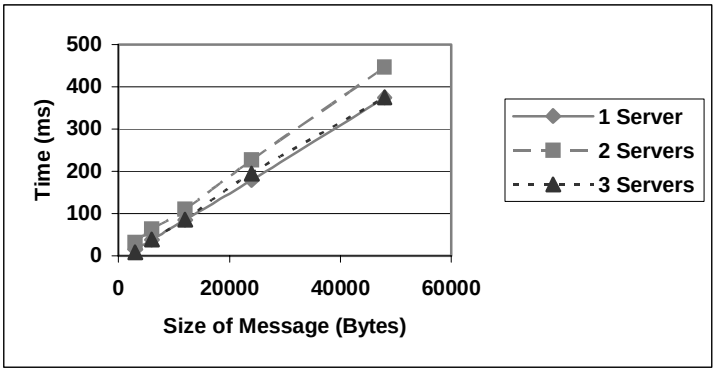

Figure 8. Result of the consolidation by quorum test.

The analysis of the obtained graphic shows that for this type of consolidation the cost for 2 servers is bigger than for 3, since with only 2 servers, all 2 have to give a reply so that the consensus can be done. With 3 servers, only 2 of them (66%) have to give the answer so that the consensus can be done (if the answers given by both are the same). A very important thing to be noted is that the time to consolidate the reply with 1 and 3 servers is practically the same, proving that the chosen algorithm is fast and efficient.

The next test shows the time that the protocol took to only send a message of variable size without waiting for a reply (one-way). Figure 9 shows the obtained results.

With this result it can be analyzed the cost of not having been used any kind of flow control in the algorithm. Messages with a considerable size (approximately bigger than 40000 bytes) start to bring a loss of packets (UDP does not have any kind of flow control like TCP), being necessary to re-send some of the messages. The solution to this type of problem is relatively easy, since the ReMIOP protocol already addresses this issue. It is interesting to note that the quantity of serves practically does not influence in the time taken to send the message, which is perfectly correct, since the message is sent only once to all of them, via broadcast.

The following tests were done to compare the ROMIOP with the UMIOP, demonstrating the cost of implementing reliable multicast and total ordering. The first of these tests show the most important reason to the loss of performance, which is the total ordering. It was done by comparing the costs of invocation of a method with reply in a ROMIOP group with the same invocation being done in a normal object (without replication) accessed via IIOP. Figure 10 presents the results.

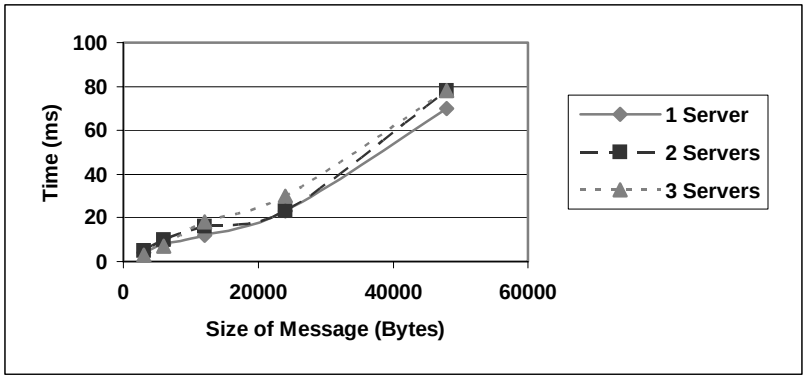

Figure 9. Result of test without reply.

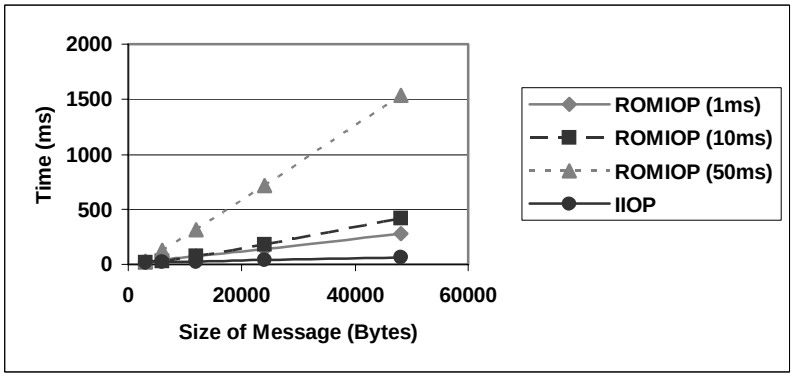

Figure 10. Comparative of the total ordering cost.

All servers must execute the request in the same order and the messages originated form older ones must be executed only after the one who generated it. To accomplish this, the servers must wait a time before sending the order of the messages to be sure that no older message will reach the servers, creating an erroneous order. This time is configurable in ROMIOP and it basically defines the protocol performance in requests where a reply is needed (two-way).

The result graph above compares the time taken to a system with only one server using a protocol that only provides point-to-point reliability and FIFO ordering (IIOP, which is the protocol that UMIOP [18, 1] uses for request that need reply) and the ROMIOP, with different times to send the message order. It is clear that any value above 10ms will slow down considerably the time taken to send the reply (look at the result with 100ms in the graph). Values bellow 10ms (look at the result with 1ms in the graph) brings very little benefits in terms of speed. This happens because the limiting factor starts to be not more the sent of the order message but the cost to accomplish the reliable multicast, creating the considerable difference to the IIOP.

Finally, the last test compares the performance of ROMIOP with MIOP [1] in requests without reply. Figure 11 shows the results obtained with 1 and 3 servers.

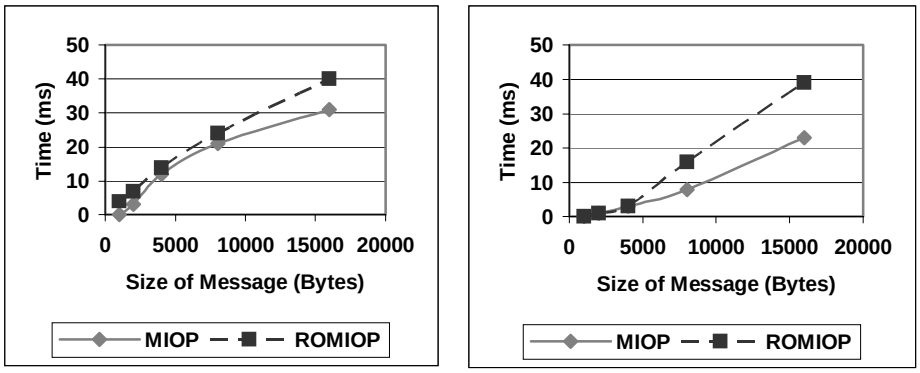

Figure 11. Comparative of protocols with request without reply with 1 and 3 servers.

The result obtained with this test revealed that for messages with small size (bellow approximately 5000 bytes) the performance of ROMIOP is really close to the MIOP [1]. Larger messages requires re-sending of some packets, making the difference grows (it is easier to visualize it with 3 servers).

7 Conclusion

This paper presented a study about the implementation of the ROMIOP draft specification using the principles of the ETF specification. The first specification aim to standardize interfaces and message formats for message multicast with total ordering and reliable multicast guarantees, while the second one defines methods for integrating new protocols into already existent systems.

One of the biggest problems we faced was the fact that the ETF specification does not support multicast communication and that it also does not easily allows to stack protocols, making the implementation of ROMIOP much more difficult and with the need to create several small extensions.

With the conclusion of this specification, we finally will have interoperable mechanisms for group communication in open (standardized) middleware. With ROMIOP now finished, it is possible to analyze each module in a detailed way, searching for a better performance, making the implementation more efficient and with more functionality.

More information related with the performance tests, the developed algorithm, as well as the source code can be found in the Internet, inside the web page of the developers group (*http://grouppac.sourceforge.net*).

References

1. Alysson Neves Bessani, Lau Cheuk Lung, Joni da Silva Fraga, and Alcides Calsavara. Integrating the Unreliable Multicast Inter-ORB Protocol in MJaco. Proc. of the 4th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems - IFIP DAIS'03, Lecture Notes in Computer Science vol 2893. Paris, France, 2003.
2. Alysson Neves Bessani, Joni da Silva Fraga, Lau Cheuk Lung, and Eduardo Adílio Pelison Alchieri. Active Replication in CORBA: Standards, Protocols and Implementation Framework. Proc. of 6th International Symposium on Distributed Objects and Applications – DOA'04, Lecture Notes in Computer Science vol 3291. Larnaca, Cyprus. October, 2004.
3. Alysson Neves Bessani, Joni da Silva Fraga, and Lau Cheuk Lung. Extending the UMIOP Specification for Reliable Multicast in CORBA. Proc. of 7th International Symposium on Distributed Objects and Applications – DOA'05. Lecture Notes in Computer Science (same volume). Larnaca, Cyprus. October, 2005.
4. S. E. Deering. Host extensions for IP multicasting. IETF RFC number 988. 1986.
5. Xavier Défago, André Schiper, and Peter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comp. Surveys, 36(4):372-421, December, 2004.
6. Pascal Felber, The CORBA Object Group Service – A Service Approach to Object Groups in CORBA, PhD. Thesis, École Polytechnique Fédérale de Lausanne. 1998.
7. Pascal Felber, Benoit Garbinato, and Rachid Guerraoui. The Design of a CORBA Group Communication Service. In Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS'96), pages 150{159, Niagara-on-the-Lake, Canada, 1996.
8. Pascal Felber, and Priya Narasimhan. Experiences, Strategies, and Challenges in Building Fault-Tolerant CORBA Systems. IEEE Transactions on Computers, 53(5):497-511. 2004.
9. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558-565. July, 1978.
10. Silvano Maffeis. Run-Time Support for Object-Oriented Distributed Programming, Ph.D. Thesis University of Zurich. 1995.
11. L. E. Moser, P. M. P. Melliar-Smith, and Priya Narasimhan. Consistent Object Replication in the Eternal System, Theory and Practice of Object Systems, 4(2): 81-92. 1998.
12. L. E. Moser, P. M. P. Melliar-Smith, Priya Narasimhan, R. R. Koch, and K. Berke. Multicast Group Communication for CORBA. In Proc. of International Symp. on Distributed Objects and Applications, pages 98{107, Edinburgh, United Kingdom. September, 1999.
13. Object Management Group. Object Management Group, Fault-Tolerant CORBA Specification v1.0. OMG Doc. ptc/2000-04-04. April, 2000.
14. Object Management Group. Reliable, Ordered, Multicast Inter-ORB Protocol. Initial Submission OMG Doc. realtime/2002-11-28. November, 2002.
15. Object Management Group. The Common Object Request Broker Architecture v3.0. OMG Standard formal/02-12-03. December, 2002.
16. Object Management Group. Extensible Transport Framework Specification v1.0. OMG TC Document ptc/2004-01-04. January, 2004.
17. Object Management Group. Reliable, Ordered, Multicast Inter-ORB Protocol. Revised Submission OMG Doc. realtime/2003-10-04. October, 2003.
18. Object Management Group. Unreliable Multicast Inter-ORB Protocol v1.0. OMG Doc. ptc/03-01-11. October, 2001.
19. Schneider, F. B. (1990) Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. ACM Computing Surveys, 22(4): 299-314. December, 1990.