Multicast-Based Interactive-Group Object-Replication For Fault Tolerance

by

Pedro Soria-Rodríguez

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical and Computer Engineering
by

April 1998

APPROVED:

Professor David Cyganski, Ph.D., Major Advisor

Professor William Michalson, Ph.D.

David Cordella, Clariion

Abstract

Distributed systems are clusters of computers working together on one task. The sharing of information across different architectures, and the timely and efficient use of the network resources for communication among computers are some of the problems involved in the implementation of a distributed system. In the case of a low latency system, the network utilization and the responsiveness of the communication mechanism are even more critical.

This thesis introduces a new approach for the distribution of messages to computers in the system, in which, the Common Object Request Broker Architecture (CORBA) is used in conjunction with IP multicast to implement a fault-tolerant, low latency distributed system. Fault tolerance is achieved by replication of the current state of the system across several hosts. An update of the current state is initiated by a client application that contacts one of the state object replicas. The new information needs to be distributed to all the members of the distributed system (the object replicas).

This state update is accomplished by using a two-phase commit protocol, which is implemented using a binary tree structure along with IP multicast to reduce the amount of network utilization, distribute the computation load associated with state propagation, and to achieve faster communication among the members of the distributed system. The use of IP multicast enhances the speed of message distribution, while the two-phase commit protocol encapsulates IP multicast to produce a reliable multicast service that is suitable for fault tolerant, distributed low latency applications. The binary tree structure, finally, is essential for the load sharing of the state commit response collection processing.

Acknowledgements

I would like to express my thanks to my advisor Dr. David Cyganski for his guidance and advice throughout the development of this research. I am also thankful for his patience with me towards the end of the completion of this thesis. I have gained a lot of knowledge and experience during my two years in the WPI Machine Vision Laboratory, and I have to thank my advisor for giving me the chance of working in this research lab.

This thesis represents the culmination of my studies thus far, and I want to express here my thanks to my parents, Pedro Soria Estevan and Maria Covadonga Rodriguez Lanza, for their encouragement and support through all my studies, while leaving the final decision to me in all the choices I had to make along the way.

My thanks also to the committee members, Dr. William Michalson and David Cordella, for their valuable comments on my thesis. I want to thank the Lockheed Martin Corporation as well, for their funding this project.

The research assistants at the MVL deserve a mention here as well, for the great work environment. In particular, this thesis would not have been completed without help from Michael Roberts. Thank you very much, Mike. Thanks to Brent Modzelewski also for explaining IGOR to me in preparation for this thesis. I would also like to thank Sashe Kanapathi. Last but not least, my thanks go also to Hsing-Yi Ko for her help throughout the development of this thesis.

Pedro Soria-Rodríguez

April 1998

Contents

List of Tables

List of Figures

Terms, Acronyms and Definitions

Asynchronous message	Non-blocking CORBA message (oneway CORBA method invocation)
BOA	Basic Object Adapter
CORBA	Common Object Request Broker Architecture
IDL	Interface Definition Language
IGOR	Interactive-Group Object-Replication
IIOP	Internet Inter-ORB Protocol (TCP/IP based)
Marshaling	Transparent data conversion between differing architectures
Object Implementation	Server object implementation of a CORBA interface
Oneway	CORBA non-blocking method invocation, also called asynchronous
ORB	Object Request Broker
TCP/IP	Transport Control Protocol layered on the Internet Protocol
Unicast	One to One communication
Broadcast	One to All communication
Multicast	One to Many communication
UDP	User Datagram Protocol
NIC	Network Interface Card
MAC	Medium Access Control
IGMP	Internet Group Management Protocol

Chapter 1

Introduction

1.1 Problem Statement

A distributed system is a group of computers working together on a specific task. This computation paradigm presents very interesting benefits over single processor systems. Distributed computing systems can render highly powerful computing engines by simply interconnecting a set of machines with otherwise very limited computational power. A good example of this is the Beowulf Project [2], a system that uses inexpensive machines in parallel to implement a supercomputer. Distributed systems also have application in fault tolerant systems, by providing redundancy. This is precisely the focus of this research work.

In general, distributed systems have to overcome problems related to the communication between the components of the system and the differing architectures of which the system may be comprised. This thesis presents a continuation of the investigation by Modzelewski [11] into implementing a fault tolerant, low latency system, over heterogeneous computer platforms.

The fault tolerance is achieved by replicating an object that maintains the state (data) of the system. A novel message distribution system was introduced in [11] for keeping all the nodes of the system in synchrony. This thesis investigates the addition of IP multicast to the system with the intention of enhancing the performance. IP multicast is used to distribute the new state to all the replicated objects by using a single message, thus reducing computation load on the part of the sender and reducing propagation delays to all the replicas. The underlying reliability mechanism is a two phase commit protocol. By using this protocol, the sender of the new state sends one message with the new state information

to the group of replicas. The second step is to gather responses from all replicas to ensure the new state has been received. Once this is done, the sender issues a commit message to the group of replicas to order them to update the old state with the new one. Once again, the replicas must reply to the sender to complete the protocol. During the operation of this protocol there are two occasions when a message must be sent to all members of the group. For this operation, multicast is used, as it achieves prompt delivery of the same information to a group of machines.

1.2 Definition of Fault Tolerance

The system addressed in this thesis is intended to supply a means for the incorporation of fault tolerance in a distributed object implementation through server object replication. The specific kind of fault tolerance which is being addressed is that which is normally associated with distributed transaction services. As stated by Coulouris, Dollimore and Kindberg [3, page 472], "A service may be described as fault tolerant if it is designed to function correctly in the presence of specified faults in the other services on which it depends. A service may be described as functioning correctly if it exhibits only the faults described in its failure semantics."

We also adpot the definition of the Lampson [9] fault model, "In order to make our assumptions about possible failures more explicit [...] we divide the events which occur in the model into two categories: *desired* and *undesired*; in a fault-free system only desired events will occur. Undesired events are subdivided into expected ones, called *errors*, and unexpected ones, called *disasters*. Our algorithms are designed to work in the presence of any number of errors, and no disasters; we make no claims about their behavior if a disaster occurs."

In Lampson [9], proof is derived that a system based upon a two phase commit protocol displays simple crash failure semantics allowing *errors* (in keeping with the definition above) in writes to permanent storage, allowing servers to crash (repeated omission failure) and allowing arbitrary delay of message delivery [3, page 460].

The two phase commitment based protocol and replicated server system developed in [11] and here, fulfills the requirements of Lampson's definition of fault tolerance given hardware failure semantics of the omission type [3, page 462].

1.3 Multicast Technologies

There are three different schemes for distributing information to other hosts on a computer network. The simplest and most straight forward is a *unicast*, by which the message originated by one computer is addressed for another, uniquely identified computer. This is a one-to-one communication. In the case that a message is not addressed to a particular host, but is simply sent for any host on the network to use, the transmission is called a *broadcast*, analogous to the broadcast by a television station. Such transmission is not addressed to a particular receiver, but rather, continuing with our analogy, any TV set can receive the signal.

In between these two schemes there is a third one, that doesn't simply transmit a message to everybody, but does not target a single host either. Such a scheme is called a *multicast*. It can be described as a broadcast addressed to a certain group of hosts. That is, there is a single message that can be received by more than one host. Another way to achieve the distribution of one message to many receivers would involve establishing a unicast with each of the receivers. However, there are important negative impacts in doing so. The main one is that the solution is not scalable, that is, the addition of more recipients of the message will increase the demand on resources in the sender. A *multicast*, on the other hand, uses a single message that is targeted at a group of receivers and hence incurs no additional costs on the part of the transmitter as the member of recipients is increased. [5]

1.3.1 Multicast Layers

Hardware

For multicast to be possible, it is necessary that the hardware interface employs a multicast mode of delivery. This is the case, for instance, with Ethernet interfaces. Each Network Interface Card (NIC) has a unique Media Access Control (MAC) address. This is the address used normally at the hardware level to deliver unicast packets. However, a NIC can also be configured to accept packets addressed to another address, in addition to its own MAC. Such an address is a multicast address. A NIC can start or stop accepting packets destined for that address at any time. The terms "joining" and "leaving" a multicast group are used to refer to the acceptance or not of packets for a specific multicast address. A "multicast group" refers to a particular multicast address. There can be more than one

multicast session active at any time. Each session uses a different multicast address. If a host wants to receive two different multicasts, it has to configure its NIC to listen on two multicast addresses.

All the NICs in the same network need to get configured to listen to the same multicast address for all of them to receive the same multicast. The Ethernet is a multiple access medium, and any packet on the wire can be seen by all the machines on the network. Therefore, if all the hosts interested in a multicast have their NICs configured appropriately, the same electrical signal that makes up a packet will cause all these NICs to receive the packet at the same time. The other NICs in the network that are not expecting the multicast will simply ignore this packet, like they do with all the unicast traffic that is not addressed to them.

This way all the hosts receive the multicast at the same time and in parallel. Physical layers other than Ethernet may distribute the multicast traffic in a different fashion. However, the relevant aspect of multicast is the fact that the originator of the message needs to send the information only once.

Software

The network layer on the hosts that are to receive a multicast also need to be configured to accept the multicast traffic. Like in the Ethernet case, the network layer has to accept packets destined for its own IP, as well as for the multicast IP. Given that any host may join or leave a multicast group at any time, the multicast IP addresses are not assigned to a particular host or group of hosts. Instead, an operation on the IP layer socket is used to specify to which multicast IP to listen. As a matter of fact, in the case of Ethernet, the multicast address of the network interface is derived from the IP multicast address.

The rest of the discussion will focus on the implementation of multicasting over IP, since this is the most commonly used network protocol.

Class D addresses are multicast addresses. These correspond to the range of IP addresses 224.0.0.0 to 239.255.255.255 [21]. A group is formed when one of these addresses is used. There is no central authority to assign addresses to groups, or to manage the membership in a group. The whole process of creating, joining and leaving groups happens dynamically.

A multicast communication is established using a UDP port at the transport level. That is, a multicast is a best effort, connection-less communication channel. There is no flow con-

trol information or acknowledgments sent back to the sender if the UDP transport protocol is used. TCP, on the other hand, provides guaranteed delivery by using an acknowledgment mechanism.

IP multicast was designed to work with UDP originally because the addition of an acknowledgment service would imply that the receivers of the multicast have to send acknowledgments back to the sender. Therefore, the sender would have to deal with a large number of acknowledgments at the same time. This would not be a scalable solution. As a consequence, rather than implementing a TCP-like multicast transport protocol, a simple UDP socket is used, along with a special multicast IP address.

An application that is to receive a multicast simply needs to wait for data on the UDP socket associated with the multicast. According to [5], the kind of applications that would benefit from multicast are:

- distributed, replicated databases.
- conferencing.
- distributed parallel computation, including distributed gaming.

Nowadays, multimedia has been added to this list. Except for the multimedia and conferencing applications, the other kinds of application usually require reliable communication. The lack of such a service has prompted several reliable multicast protocols to be developed in the last few years. These are discussed in section 2.7.

1.3.2 Multicast Internetworking

When a multicast spans more than one network the multicast traffic needs to be forwarded by the router or switch onto the next network, since these devices interconnect two or more different physical layers. The router has to deal with the problem of routing the multicast traffic only to those networks where there are hosts that are expecting the multicast. Similarly, it does not need to send the multicast to a network where those packets are not needed.

The solution is to inform the routers that there are hosts expecting a multicast in the networks connected to them. The Internet Group Management Protocol (IGMP) is the protocol used to discover the group membership in a network. The routing of multicast over

the internet can be done with a variety of protocols that can be classified into two main groups: *dense-mode* and *sparse-mode* protocols. The first type of protocols is intended for large networks that are largely populated with members of a multicast group. These protocols are Distance Vector Multicast Routing Protocol (DVMRP), Protocol-Independent Multicast - Dense Mode (PIM-DM), and Multicast Open Shortest Path First (MOSPF). *sparse-mode* protocols are used for networks in which not many subnets may contain multicast group members. Some of these protocols are Protocol-Independent Multicast - Sparse Mode (PIM-SM) and Core Based Trees (CBT). More information about these internet routing protocols can be found in [15].

An IGMP message is encapsulated in an IP packet, and has a time-to-live (TTL) of 1, to avoid it being forwarded onto the next network [4]. The router sends a query to address 244.0.0.1, the address of the all-host group. When a host receives the query, it replies with another IGMP message, addressed to the multicast IP address of the group whose membership it is reporting. It sends one such message for each multicast group to which it belongs. A host waits a random amount of time before sending a reply, so as to avoid an implosion of replies from all the hosts on the network. Should there be more than one member of one group on the same network, only the first membership report generated would reach the network, because as soon as one host reports its membership, the other members of the same group would see that message (being a multicast) and would abort their respective reports of membership. One message is enough for the router(s) to be aware that there is at least one host on a particular multicast group. The report messages also have a TTL equal to 1. Routers post the IGMP queries periodically on their networks. If after a few consecutive queries there are no responses to the polls, the router assumes that there are no host group members on that network.

Not all routers are multicast-capable, although a lot of new products implement IGMP. Vendors explicitly indicate whether their products support IGMP or not. Older routers that are in place in the internet are less likely to support IGMP.

Chapter 2

Reliable Multicast

In the field of distributed computing there is a need for reliable communication. A multimedia application can usually tolerate the loss of a certain portion of packets. The information content in the video or audio stream may not suffer a high degree of quality loss. On the other hand, a distributed computing application cannot extrapolate the information of a lost packet from another data packet. It is therefore necessary that the transport layer provides reliable service.

This chapter presents background information about the problems that a reliable multicast service needs to address. Along with the problems, potential solutions investigated throughout this research are presented, as well as a summary of other work developed in the area of reliable multicast.

2.1 Problem Description

This section presents a discussion of the difficulties that have to be dealt with in an acknowledgment-based multicast system with retransmissions. In its simplest description, such a system requires that the receivers of the multicast inform the sender, through an acknowledgment frame, that the packet has been received. The sender will retransmit a packet for which an acknowledgment has not been received in a certain period of time. An acknowledgment is required from all hosts in the membership of the group.

Let us consider a multicast host group whose members are physically located far from each other and look at the problems that arise from this situation. For instance, if the farthest member of the group (from the sender's point of view) has missed a frame and

requests a retransmission, the request would have to travel the longest distance within the group topology, and the retransmitted frame would have to propagate back down the same long path. In addition to suffering from large delays in discovering lost frames, this method puts a big load on the originator of the message, since it must process retransmission requests from all the members of the group.

One way to reduce the latency and the load on the sender would be by having the group member that did not receive a certain frame contact another member of the group (a closer one to it) rather than contacting the original sender.

Another possible scheme for requesting retransmission would be to not send any kind of acknowledgment back from the clients (or receivers) to the sender. This scheme would be based on the premise that a good number of frames (more than 50%) do not get corrupted nor lost. If this was true, a client could detect when it is missing a frame by checking a sequence number on every frame. This way it would only be needed to generate a "NACK" message (negative acknowledgment) from the receivers to the sender. This would allow it to avoid sending an acknowledgment for every frame that has arrived correctly. That is, only in the event of a lost frame, a NACK is sent.

In either case, if the client that is missing a frame is the one farthest from the sender, the delay caused to the overall communication would be greater than necessary given more local stores of information exist.

2.2 Retransmission

Let us now consider the problem of retransmission in more detail. As stated previously, a good solution would be to request the lost frame from a group member that is closer than the source of the multicast. Each group member can maintain a table with information about the other members, that can be updated periodically.

The problem of avoiding a client's having to contact a distant server to request the retransmission of a packet can be solved if the client can contact a closer source for the information it requires. Given that there are, most likely, other members in the same group, some of these members may be closer (in distance or in round-trip time) than the original sender. All of these host members are receiving the multicast as well. A mechanism can be set up to have the clients contact another client to request the missing frame.

This implies that a client needs to know about its own location with respect to its

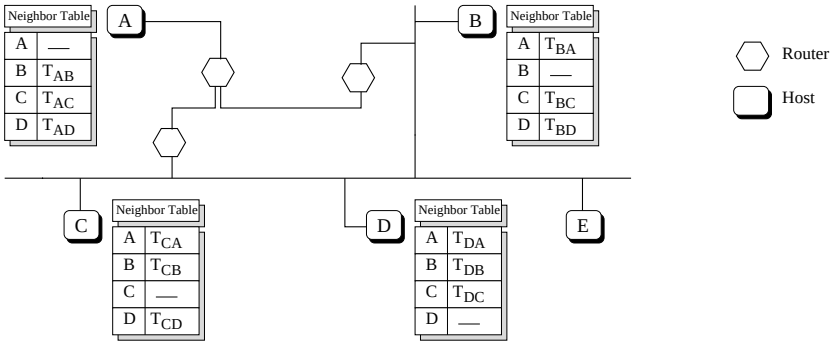

Figure 2.1: Neighbors Tables

neighbors (other members of the group) and to the sender. If such information is available to every client, each client can then contact the closest neighbor to request the missing information. This method of implementing retransmission requires an efficient scheme to provide each client with the means to find its place within the group. A simple method to realize this would involve the measurement of the round-trip time of frames sent to each other member of the group. This would result in a table that lists the closest neighbor(s) to each member. The optimum time to generate this look-up table would be at the time when a client joins the group. Figure 2.1 shows a group of hosts on different networks, connected through routers. Host "E" is not part of the multicast group. The other four hosts are, and maintain a table with the round trip time to each other, T_{XY}.

Now let us suppose that a new group is set up and there is a number of hosts that wish to join this group. If all hosts start transmitting frames to collect data for the *neighbor tables*, a sudden burst of frames will appear on the network, both incoming and outgoing for each host. This amount of traffic might affect the computation of the *neighbor table* and generate erroneous results, which in turn will lead to performance lower than what would otherwise be possible.

However, if a host that wishes to join a group employs some mechanism for introducing a small random delay before its transmission, this problem can be lessened. Using a scheme similar to this, the negative impact on network load imposed by the set-up phase will be reduced to a minimum. These *neighbor tables* can have three important uses:

1. Maintain information about the distance (in milliseconds, router hops, etc..) between

members in the group.

2. If these tables are updated with certain periodicity, then the messages used to gather the distance information become also a heartbeat signal to maintain a record of membership in the group.

3. By requesting that a new host joining the group build its own *neighbor table* when it first joins the group, this mechanism is also a way for a new host to advertise its presence to the group.

2.3 Multicast Subgroups

An alternative to contacting individual clients is to request the retransmission of some frames from one of several "servers" within the group. These servers would then provide reliable service to a subgroup of the members of the greater group. The original sender could contact the servers for each subgroup either using a unicast or a multicast, depending on the number of subgroups that it would be worth creating. It is implied that the number and size of these subgroups will vary depending on physical location of the hosts, number of hosts on one side of a router and so on.

Formation of subgroups within the multicast group involves that all members of the group must have some way of knowing about their physical location with respect to each other, as was needed in the previous scheme of each host contacting the closer neighbor. The algorithm to create the subgroups has to be very flexible to allow, for instance, that the main sender of a subgroup can leave the multicast group if it so desires.

The path of communication between the servers of the subgroups and the surrounding clients is shortened with respect to a big multicast group with one central distribution machine. Given the path is now shorter, the probability of error in a multicast frame can be lowered, and thus we reduce the number of NACKs generated as well. The main server of the group could communicate with each server of the subgroups through TCP to ensure that this first communication is established without errors. The server in each sub-group then might use multicast to broadcast the frame to all clients under its umbrella.

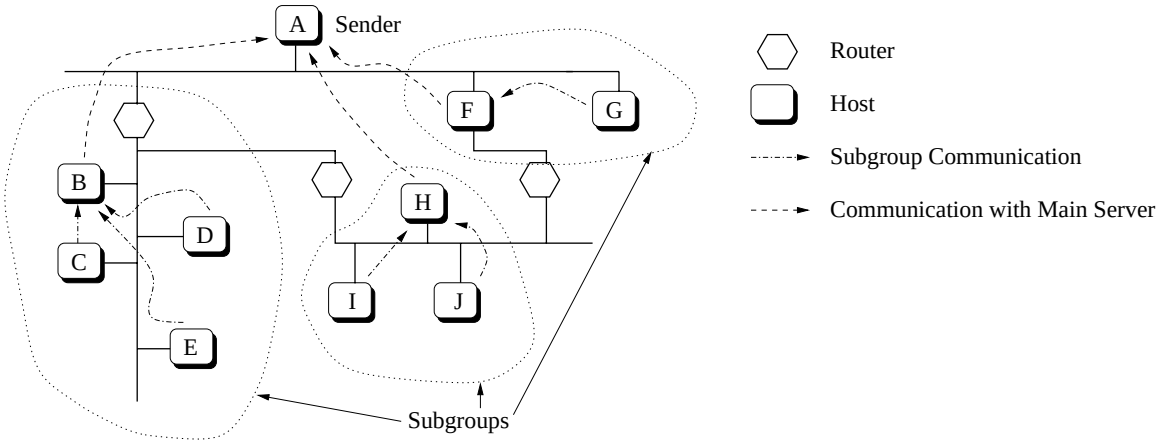

Figure 2.2: Multicast Subgroups

2.4 Keeping track of group membership

In order to establish a reliability mechanism based on acknowledgment messages, it is necessary to specify a system to maintain an up to date record of members currently in the group.

We will explore means to do this by using a data structure that is continuously shared by all members in the group. This way every member can make their presence clear to the group. In the same way, the rest of the group can know if a given host is still a member of the group or not. The main advantage is that this mechanism is practically de-centralized. There is still a need for a main server of the group, but the group does not depend on it for operation to a critical extent. This is how it works:

1. There is a table called the "membership table", which contains a list of all the members currently in the group. Along with each entry there is a counter for each host, which contains a time-to-live (TTL) value. This table is generated at the main server. The server passes it to the next host (member of the group) in the table. This one, in turn passes it to the next host, and so on. The table eventually reaches the server again, when it has been passed through each of the members of the group.

   When a host receives the table, it decrements the TTL of all the other hosts, and updates its own TTL to the initial value. Then it passes the table to the next host

in the list. A host who's TTL in the table has reached zero will be dropped (deleted) from the membership table by the next host that discovers that the TTL value was zero. A host that is in the group will be continuously updating its TTL value in the table. This is the proposed mechanism to keep all hosts informed near-simultaneously about the membership of the group.

The operation of this method is illustrated in figure 2.3. Figure 2.3.1 shows that host "A" currently has the membership table after the table was initialized with the default value of 9 for all hosts. Host "A" has decremented the count for all other hosts and kept its own count at 9. In Figure 2.3.2 the table has been received by member "B", who has updated its own count to the default value and it has decremented the count for the other members. The same process takes place in the next two steps. In 2.3.5, however, the table has completed a cycle around the group membership and has returned to host "A". It can be seen that host "A" has updated its own count once again, and it has decremented the counts of the other group members.

2. When a new host wants to join the group, it sends a message to the main server to request membership in the group. The server then adds this host to the membership table, and sends a new table to the next host in the list. When the server receives the old table again, it will discard it, by looking at a table sequence number that identifies which table revision is the current one. The server host is the only one that can destroy a table and create a new one.

3. If a host dies, it will be first noticed by the host that precedes it in the membership table (host A). When attempting to connect to the (dead) host, host A will give up if the connection attempt results in error. It will then pass the table on to the following host in the table (the one after the dead host). If the host that was not responding was in fact down, it will eventually be dropped from membership when its TTL value reaches zero. If the connection did not take place the first time because of some transient error, this host will have a chance to increment its TTL again the next time it receives the membership table.

4. In the event that a host wants to leave the group, it simply has to wait until it gets the membership table. It can then delete itself from the table and pass the table to the next member.

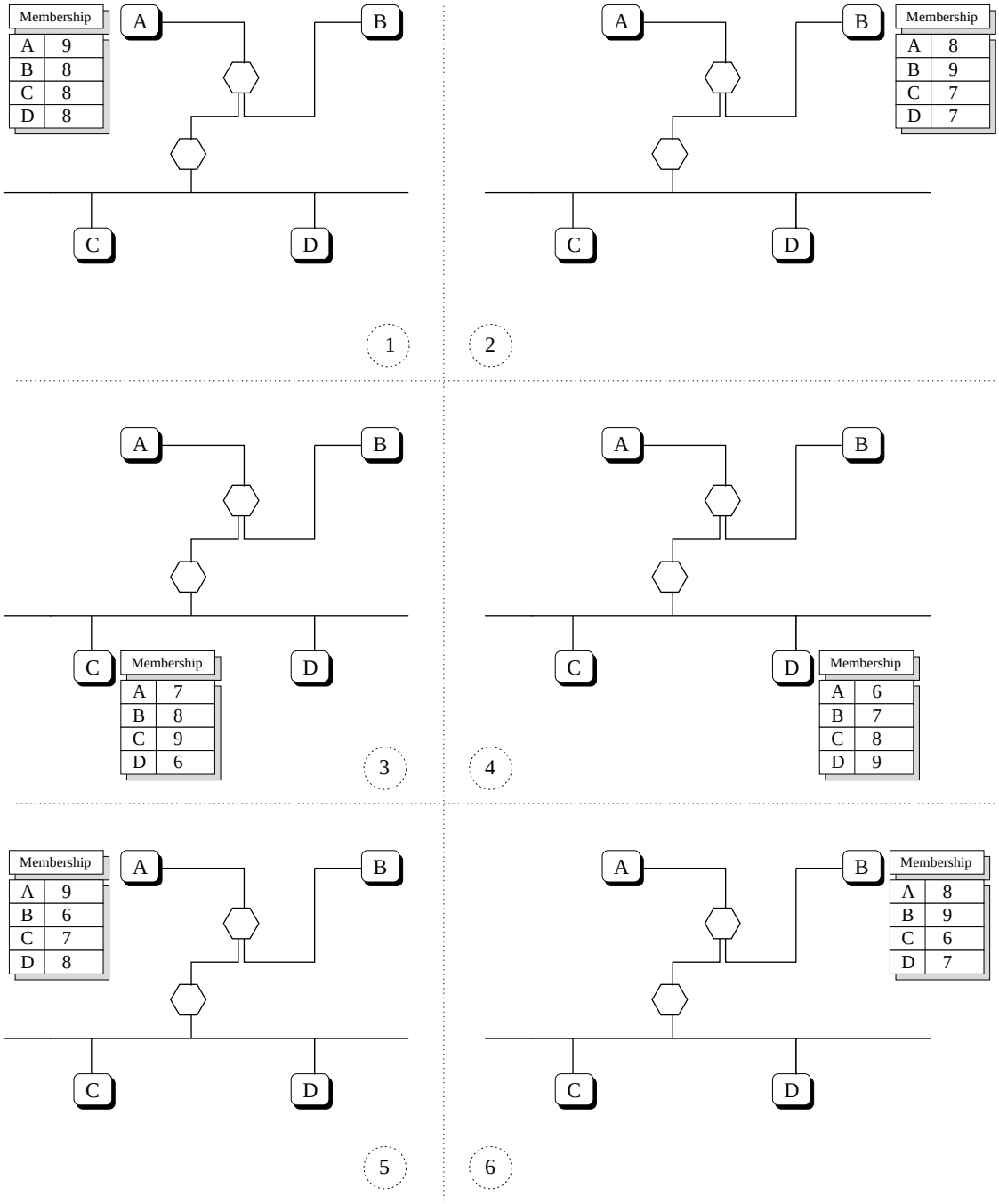

Figure 2.3: The updating of member tables

5. As soon as a host receives the membership table, it updates its own table with the new information, so as to know how many acknowledgments to expect from the other members when it sends a multicast.

6. The membership table can contain other information about each host. For instance, a valuable piece of information to know would be who within the group is willing to become a server for the group, in case the original server dies for some reason. This could be in the form of a flag next to each entry in the table indicating that a host is willing to become a server of the group. If the host that is supposed to pass the membership table back to the original server finds that the server does not respond, this host will look in the table for a machine that is willing to become a server and send it to that one.

7. It is the job of the group server to send the membership table around the group periodically. A period of 25 seconds could be appropriate to discover dead hosts in many DBMS applications. The server generates a new membership table when a new host joins the group, so there is no delay in communicating the presence of a new member to the whole group. The periodicity of the transmission of this membership table makes it a sort of heartbeat signal, similar to that found in other reliable multicast protocols.

8. The protocol used to pass the table on to the next host could be TCP, since it is a one-to-one communication. By using TCP we also make sure that the table is being passed to the next host, and we can detect if the next host is down or not responding.

2.5 Retransmission Request

If the multicast group spans large distances and a delivery error occurs, the host that needs a retransmission needs to request the retransmission from the sender of the missing frame. This sender may be relatively far away from the receiving host. The time it would take to send the request and the new message back and forth over such long distance, perhaps through many routers, may be too long to be practical for a given application. Another problem this imposes is that, in the case of a multicast retransmission, the new transmission would be seen by all members of the group, even though this is not necessary. In a more efficient design, we will try to avoid these inconveniences.

One solution is to request the retransmission from another host, different than the original sender. Since all members of the group are receiving the same multicast, any of them has a copy of the message that we are missing. This problem brings the issue of how to know which hosts are closer to us.

A possible solution to this problem is to maintain a look-up table in each host with the distance to all other hosts. The distance might be stored in units of time (a round trip time (RTT) measure) or in router hop counts.

To gather this information, each host needs to send a message to each of the other hosts in the group. This is a special message that will ask the other host to send another message back. This way we can compute the round trip time to that host. A table is built with an entry for each member in the group. This needs to be done only once, at the time when a host joins the group. When a host joins the group it will send this special message requesting another message of the same type back. When the other hosts in the group receive this RTT-measurement request they know that there is a new host in the group and they can send the same kind of request to the new host, by piggy-backing it in the reply they send to the new host. The new host then replies back to all those other hosts. Now all hosts have a table that lists the distance to each other. This distance table is updated every time a host receives a copy of the membership table. If a member has dropped from the group, it can now be erased from the distance table. If for some reason a new host did not request distance information from us, and we receive the membership table and see a new host, we can add it to our distance list and request the distance information from it.

Figure 2.4 shows the three steps of this process. In Step 1, hosts “A” and “B” do not know yet about host “C”. For this reason, their tables do not include “C”. Both “A” and “B” have previous knowledge about the RTT with respect to each other. In Step 2, hosts “A” and “B” have sent their responses to the RTT request by “C”, and therefore “C” can compute the RTT for the two existing members of the group, and put the values in its table. After Step 3, both “A” and “B” receive the response to their RTT requests, and complete their membership tables with the RTT to host “C”.

This table can then be used to find the host closer to us who is also willing to be a server. This information was stated in the membership table. When we receive our copy of this table, we can look for potential servers in the group and copy that information in our distance table. The distance table could then be reduced in size to accommodate only entries for hosts that can work as servers of the group.

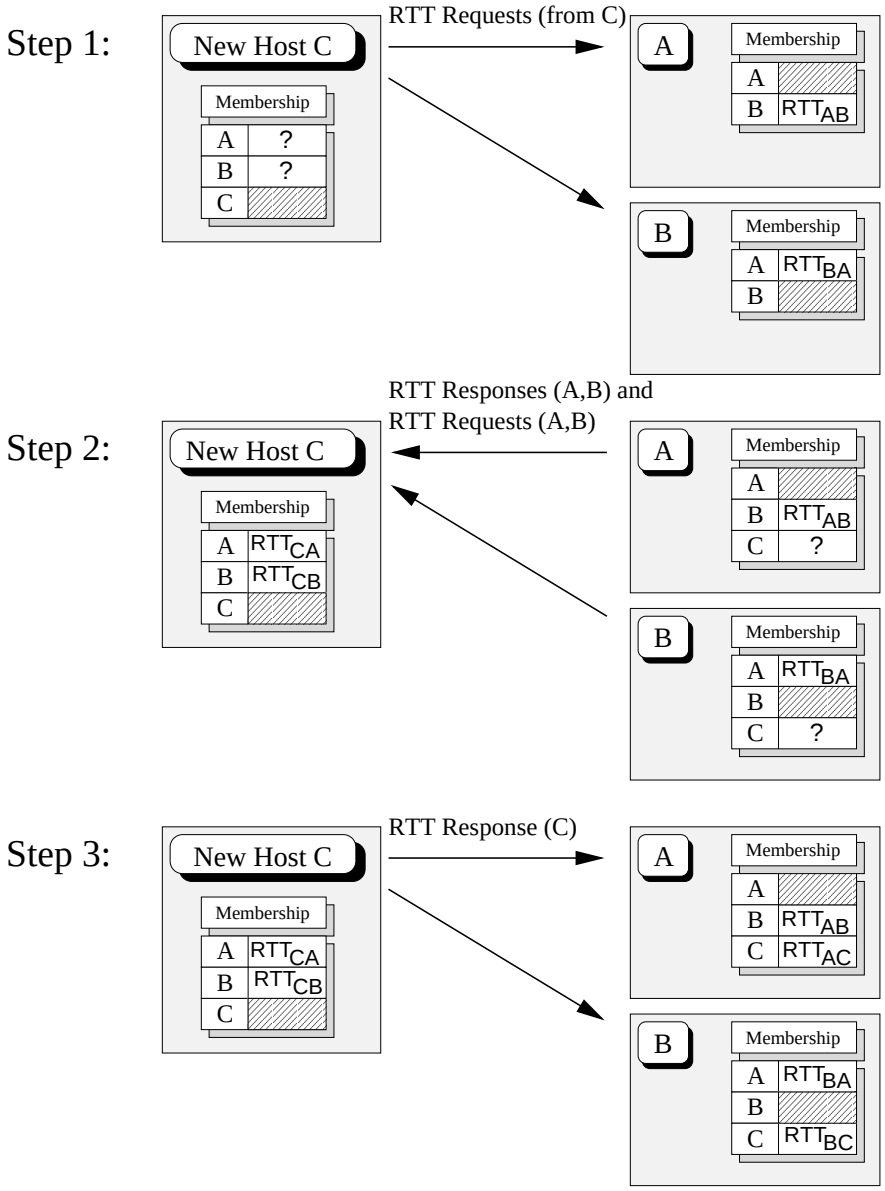

Figure 2.4: Computation of Round Trip Times (RTT)

If the retransmission request is performed using multicast, we can limit the time to live (TTL) parameter of the IP frame, so that it does not reach beyond the server with which we are trying to communicate. For this reason, it would be a good choice to store the distance information as a hop count rather than a measure in time. The server will then use the same TTL to multicast the retransmitted frame. This way it will not reach all the hosts in the group, thus reducing the number of acknowledgments that are generated.

2.6 Other Issues

There are other considerations to take into account when designing a reliable multicast system. Some of the following features can be found on the transport control protocol (TCP), and would be necessary to produce an efficient multicast transport protocol.

- How to implement a sliding window algorithm in a multicast environment and how big it should be.
- Whether or not to implement flow control, and how.
- Investigate potential applications to determine speed requirements.

2.7 Previous Work on Reliable Multicast

While there are some reliable multicast implementations dating back to 1984, there has been a lot of activity in this field in the past few years. The impressive growth of the internet and the new *intranets* have opened the way for applications that can greatly benefit from multicast.

An overview of some significant works about multicast is presented in this section.

2.8 RMTP: A Reliable Multicast Transport Protocol

RMTP provides sequenced, lossless delivery of bulk data from one sender to many receivers [10]. The ACK-implosion problem is avoided using a hierarchical structure that divides the receivers in subgroups. A designated receiver in each group is responsible of retransmitting lost packets within that subgroup. RMTP builds on top of a best-effort network layer like IP.

Assumptions in RMTP

This protocol focuses on providing reliability, scalability and heterogeneity. What is meant by reliability is that all receivers receive an exact copy of the file transmitted by the sender. Reliability is achieved by periodic transmission of status by the receivers, and selective-repeat retransmission. That is, retransmissions are sent only to those hosts that missed a certain packet.

In case of network partition, RMTP does not provide reliability, but it notifies the application about the fault. A host that leaves the multicast group is not guaranteed reliable delivery either. It is assumed that there is a Session Manager to handle the group membership. The responsibility of ensuring lossless data reception is placed on each individual receiver.

A windowed flow control with congestion avoidance is used to avoid overloading slow receivers. The data rate is one of the parameters assumed to be controlled by the Session Manager .

Reliability Mechanism in RMTP

Certain members of the multicast group called ACK Processors (AP) are responsible for retransmitting lost packets as a multicast or a unicast, depending on the number of receivers that lost the packet. There are several APs in the group. Each receiver reports lost packets to its corresponding AP. This way, the multicast group is divided in subgroups to avoid the ACK implosion problem.

The hosts that act as AP are selected ahead of time, presumably by the Session Manager. If a host acting as an AP leaves the group or gets disconnected from the group for any reason, RMTP provides mechanisms for the receivers to contact another AP in order to obtain their retransmissions. However, APs cannot be chosen dynamically if the membership of the group changes.

Members can join and leave the group and still receive data reliably.

2.9 Reliable Multicast Framework

This paper describes a protocol called Scalable Reliable Multicast (SRM), a reliable multicast framework for application framing and light-weight sessions. The authors argue

that multicast applications have different requirements in the type of reliability needed, and for this reason the design of a reliable protocol should not follow the "one-size-fits-all" paradigm [6].

The reliable protocol described here is tested with a network conferencing tool created by the authors, called "wb" (whiteboard). It is a shared whiteboard where all the participants can write and read. This is a many-to-many kind of multicast application.

When new data is generated by the whiteboard, it is multicast to the group. Each member is individually responsible for detecting errors in the reception, and requesting retransmission. The detection of losses is done by inspecting the sequence number. However, the last packet of a transmission may have been lost and the receivers do not know that they should have received this last packet. For this reason, receivers advertise periodically the highest sequence number they have received so far from every other member of the group. These session messages include time-stamps that are used to determine the distance (in time) from each other member itself.

The distance calculation is used to estimate a random amount of time to wait before a member sends a request for retransmission. This way, if two receivers missed a packet, but one of them is closer to the source of the packet, the closer one will request retransmission before the second one does. Thanks to this mechanism, the two hosts that missed the packet do not request retransmission. Only one of them does, but since the retransmission is done as a multicast, every group member benefits from the retransmission. This prevents the request-implosion problem.

There is no mention of subgroups within the multicast group. In this protocol every member is a sender, and therefore every member has to ensure reliability from all possible sources of information. A network partition is not different from a member leaving the multicast group. There is no need to keep track of the membership of the group since the reliability is receiver-based rather than sender-based.

2.10 Single Connection Emulation (SCE)

The authors in [20] propose to introduce a new layer in the OSI model, called the SCE layer. This layer implements a protocol called reliable multicast transport service (RMTS). This is another one-to-many protocol. One of the assumptions is that there is a single source and many receivers.

Under this protocol it is possible to drop out of a multicast connection, but not join an existing one. When a host drops from the multicast group it stops sending ACKs back. After a certain period of time without receiving ACKs, the sender considers this receiver to be out of the group.

The SCE layer acts between the transport and network layers. Connections made from the transport layer pass by the SCE layer, which passes the connection request to the network layer. The network layer uses a multicast protocol (IP Multicast, for instance) to multicast the data. The SCE then gathers the ACKs received from all the members in the group. When it has received all the expected ACKs, it send a single ACK to the transport layer, simulating a point to point connection from the point of view of the transport layer.

Instead of modifying existing protocols, this paper proposes a new layer that fits in with existing layers, by using the existing interface to the other layers, and adding functionality in this new layer.

2.11 Reliable Multicast Protocol. (RMP)

This protocol provides many-to-many multicast communication. Different levels of quality of service can be specified. All members play the same role in the communication. RMP uses a combination of positive and negative ACKs. In RMP, the group is organized as a ring. The ACKs are passed as a token between all the members, thus distributing the load of processing the ACKs.

Each member of the group that is sending information to the rest of the group must keep a cache of the data in case it needs to be retransmitted. In a reliable communications protocol a message is said to be stable when the sender of the packet knows that all targeted destinations have received the packet. A stable packet no longer needs to be held for retransmission. RMP notifies the senders when their packets have become stable.

2.12 The IP Multicast Initiative

Lastly, it is worth mentioning the *IP Multicast Initiative* [16], a join effort by a large group of companies to promote the use and demand for IP multicast. This organization provides white papers and technical reports about IP multicast through their web site, and sponsor conferences and events relating to IP multicast. This organization is a proof that

the industry interest in multicast communication is evident.

Chapter 3

IGOR

The **I**nteractive-**G**roup **O**bject-**R**eplication system [11, 12] is a framework for achieving fault tolerance in a low latency system by introducing redundancy in the object implementation that maintains the current state of the system. This mechanism provides fault tolerance against host crashes or network partitions. In the event that one of the copies of the database object (also called replicas) crashes or cannot be reached, the rest of the replicas can still provide the service independently.

A client trying to retrieve or update the state in the database object can do so without knowing about the replication of the database. It is not necessary for a client application to contact one specific replica. From the point of view of a client, there is simply an interface from which to obtain or modify the state of the database object. The replication mechanism is in charge of ensuring that all the replicas of the object get updated with the new information when the client modifies the state in one of them.

Another flexibility in the design of **IGOR** is that the number of replicas can be variable. More can be added to the existing group of objects dynamically, without a need to reconfigure the system. Upon joining the group, a new replica receives a copy of the current state and it is then ready to take requests from clients. Similarly, replicas can leave the group without disruption of service.

In order to add platform independence to the design, IGOR is based on CORBA. CORBA is a middleware service that allows for the inter-operation of software running on different operating systems and different hardware platforms. An object running on one machine can invoke methods in another object that may be running on another machine. CORBA handles the conversion of data marshaling among differing architectures.

3.1 IGOR Implementation

IGOR is intended to be a layer that can be added to an existing application. By simply running several copies of the application, the fault tolerance is achieved without further intervention on the part of the administrator of the application.

In the IGOR system there is an additional process, called the *registry*, that keeps track of the number of replicas that exist. Another task of the registry is to inform all the replicas of the new membership of the group when new replicas join, or when a replica has failed.

When a client connects to one of the replicas to change the state, the replica first propagates the new state, using a two phase commit protocol. Once all the replicas are synchronized, the one that received the request from the client replies back to this one. The distribution of the information inside the group has to be done quickly in order to have a responsive system. At the same time, the solution has to scale well given that an application may require a large number of replicas.

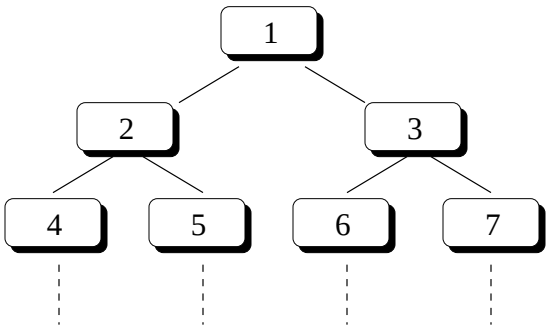

Figure 3.1: IGOR Binary Tree

For these reasons, it would not be a good solution for the recipient of a client request to simply contact every replica individually and pass the new information on to them. The approach taken in IGOR uses a binary tree to structure the replicas in the group. When a new replica is created and joins the group, it contacts the registry to inform it of its presence. The registry adds this new replica in a binary tree of replicas, and it tells the newly added replica of its position within the binary tree. In the binary tree in figure 3.1, a new replica could get added as the left child of replica number 4, for instance. The reason for the binary tree is that if the message is propagated from the replica at the root

of the tree by way of unicast transmissions from each node on the tree to its immediate descendants, the time required to get the message to all the replicas is much less. Each replica only needs to propagate the message to its two children, who will in turn propagate it to their two children, and so on. This implies that the propagation of the message can be done in parallel from the two children of the root of the tree, and the same will happen for the children of every node in the tree.

The time required to propagate the new message (from the root of the tree) to a group of N replicas is $(T_{msg})log_2(N)$, where (T_{msg}) is the time required to propagate one message from one replica object to another. When compared to the total time necessary if one replica has to contact all others $((T_{msg})N)$, there is a clear advantage in using the binary tree approach.

3.2 Two Phase Commit Protocol

To ensure the consistency of the data across all the replicas in the group, a protocol called the *Two Phase Commit Protocol* [13] is employed. It is used to make all the replicas change their state consistently, and to ensure that none of the replicas misses the new data. If the protocol fails, indicating that there is not consistency among all the members of the group, the state change operation is aborted. The replica that received the client request contacts the client to tell it that the operation was not completed.

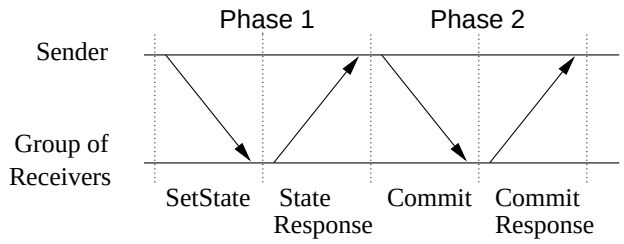

Figure 3.2: Two Phase Commit Protocol

The operation of this protocol is summarized in figure 3.2. In the first phase, the replica that initiates the propagation of the data sends the new data to all the replicas (*SetState* call). When the other replicas receive this message, they store this new data in a temporary location, without replacing the old data. At this point they reply to the sender, with the

SetState Response message. Upon collecting responses from all the replicas in the group, the first phase of the protocol is finished. The second phase begins when the original sender sends a *Commit* message to the whole group. When the other replicas receive this message, they are sure that every other replica in the group has received the new data, and can now change their old data to the new one. They then issue the *Commit Response* message. When the original sender has received all the responses from all the replicas in the group, it is in a position to reply back to the client and confirm that the new data has been stored. From the client's point of view, there is no knowledge about the replication of the data. It only needs to know whether the state was saved correctly or not.

Failures can occur during the first phase, if one of the replicas did not receive the *SetState* message, or if its *SetState Response* message was not received by the initiator of the protocol. If the first phase fails for any of these reasons, the original sender will issue a *Rollback* message to abort the operation. Since all the replicas do not change the state immediately, they can discard the new data and keep the old one when they receive this message.

This is the normal operation of the two phase commit protocol between one sender and one or more receivers. In our case we are using a binary tree to distribute the messages more efficiently, and therefore the message is not arriving to all the replicas at the same time. Furthermore, a node in the tree cannot send a reply message directly to the sender of the original message, but it has to respond to its neighboring nodes first. Therefore, the two phase commit protocol requires special implementation when using the binary tree structure.

Let us assume that the replica that receives the client request to change the data is at the root of the tree. This node will perform the *SetState* call on its two children. In turn, these replicas perform the same call on their children, and so on. Next, a replica does not reply back up to its parent until it has received the *SetState Response* messages from both of its children, if it has any. Therefore, the replicas at the lowest level of the tree will initiate the response because they do not have any children. When their parents receive these responses, they can then reply to their parents, and so on, until the responses reach the root of the tree. This concludes the first phase of the protocol. The second phase works analogously.

IGOR presents an easily scalable solution for distributing a common message to a set of replicas. The fault tolerance is achieved by object replication and by blending the two phase

commit protocol into the binary tree structure. As mentioned earlier, part of the reliability in IGOR is due to the use of CORBA as the communication system between replicas. The CORBA implementation used contributes some of the fault tolerant features of IGOR. For instance, it is possible to have an object automatically be restarted if it suffers a crash. If an object A attempts to communicate to another object B which is not running anymore, CORBA lets A know that object B is not available. [23, 22]

3.3 Enhancements to IGOR

The central problem that IGOR tries to address is the addition of fault tolerance. This is accomplished through replication of the database object. However, this translates into an implementation problem for fast communication among replicas. IGOR solves this problem by structuring the replicas in a binary tree.

There is an alternative to the binary tree solution: to use multicast for distributing the messages, as noted in [11]. By switching from a unicast system to a multicasting one, the need for the binary tree disappears. If the message indicating the update of the information is sent by multicast, all of the replicas will receive it literally at the same time, except when delays are introduced by routers. The binary tree was used to reduce from N to $log_2 N$ the time to propagate the message to a group of N replicas. With multicast, the reduction factor a constant K instead.

While multicast can be used to distribute messages to the group, it may not be appropriate for the responses from the group back to the sender. If all the replicas were to respond with a multicast, they would all do it at the same time, since they all receive the first message at the same time. This implies that the original sender, who is collecting the responses from all other replicas, would have to process $N - 1$ incoming messages. This solution does not scale.

Therefore, it is more appropriate to use the binary tree structure to collect the responses as they progress up the tree. With this approach, one replica has to process at most two responses, from its two children. This will yield a total time to collect all responses of $(T_{msg})log_2 N$. The use of multicast for this task would require a time of $(T_{msg})N$ at the replica receiving the responses to process all of them. In addition to this, the implosion of responses could cause buffer overflows and collisions on the network, thus making the process even slower.

In summary, it is feasible to implement the multicast to distribute messages, in both the *SetState* and *Commit* calls. For the corresponding responses from the group of replicas (*SetState Response* and *Commit Response*), the tree can be used to collect the responses.

3.4 Multicast for Fault Tolerance

As described in section 2.1, multicast does not provide a reliable service like TCP. Consequently, the choice of multicast for a fault tolerant application may seem like a wrong one. One of the reliable multicast protocols described in section 2.7 might be used as the replacement for unicast (TCP). It would ensure the same reliability as TCP, but the advantage of plain IP multicast would be lessened. All the reliable multicast protocols must provide the reliability at some cost over simple multicast. Even so, reliable multicast would be a more efficient and scalable solution than unicast.

On the other hand, the need for reliable multicast in the IGOR framework may be unjustified. IGOR bases its fault tolerance and reliability both on CORBA (using unicast TCP) and on the two phase commit protocol. The two phase commit protocol comes into play when there has been a failure in the TCP delivery mechanism. In the case that the message distribution mechanism is multicast, the two phase commit protocol is still in place to handle failures in the transport protocol.

3.5 MIGOR

MIGOR is the name given to the Multicast-enabled version of IGOR. The architecture of MIGOR is very similar to IGOR.

The application object inherits the fault tolerant characteristics from a MIGOR object. The `MIGOR` object keeps knowledge of the binary tree structure, and it also handles the transaction processing operations defined by the two phase commit protocol. MIGOR in turn inherits the multicast capability from another class called `Multicast`, which is derived from `McastModule`. This last one implements the actual operations for joining and leaving multicast groups, and sending and receiving data through a multicast port. The class `Multicast` is a CORBA wrapper around the `McastModule` class to provide a CORBA interface for the use of multicast. Presently this class is quite simple and does not implement the CORBA interface yet. However, the addition of the CORBA interface would be placed

on this class, and not on `McastModule`. This way it is possible to modify the implementation of the multicast functionality without affecting the rest of the modules, since the interface would remain the same.

Chapter 4

Multicast IGOR (MIGOR)

The focus of this thesis was to design, implement and test a version of IGOR that uses multicast in order to compare its performance with the original work developed in [11].

4.1 Choice of ORB

In [11], the Visigenic (now Borland) ORB (Object Request Broker) called *Visibroker* was used for the insertion of CORBA into IGOR. It is a CORBA 2.0 compliant ORB, with some additional features. One of these is the *naming service*, which is not part of the CORBA 2.0 specification but *Visibroker* includes a proprietary implementation of a naming service. A naming service provides the ability to find an active object implementation in a network given an identifier. It is possible to specify only the type of object it is (i.e., its interface name, as specified by the IDL) or the interface name along with the name of a specific object. The naming service is used when an object **A** (of interface type `interfaceA`, say) needs to obtain a reference to another object **B** (with type `interfaceB`). The location of the object **B** (the computer and TCP port it is running on) is unknown by object **A**, and this is precisely the problem that the naming service solves. The naming service provides **A** with the information it needs to contact **B**. In order to do this, **A** has to do a `bind` operation specifying the interface type of the object of which it wants to get a reference. It can optionally indicate a particular name. The naming service uses this name to locate a particular object, in the case that there are several active implementation objects of the same type of interface.

IGOR makes use of it for allowing the new replicas to automatically find the registry

service on the network without user interaction.

By using the same ORB in MIGOR as well, its performance can be compared to the IGOR system more accurately.

4.2 MIGOR Overview

The MIGOR system consists of three main entities: a *R*egistry Service, a *c*lient and *o*ne or more servers. The servers are the different replicas of a single database object. The registry is in charge in charge of keeping track of the number of replicas present. It also provides the replica objects knowledge about each other. Finally, the client is the application that retrieves data (current state) from the servers, or makes changes to the state.

4.3 Registration of Replicas

When a new replica is started, its first task is to obtain knowledge about other replicas that may exist already. Rather than sending a broadcast-type of query to find other replicas on the network, it contacts a special server, called the *r*egistry. Thanks to the naming service in *Visibroker*, it is possible to find a CORBA object on the network given the name of its interface. This interface refers to the IDL-defined interface.

The registry's IDL interface name is `RegInterface`. Through a CORBA operation called `bind`, the new replica obtains a pointer to the replica object. That is, it gathers information about the host where the registry is running and how to contact it. By performing a call on a method that the registry provides (called `AddReplica`), the replica registers itself with the registry object. This is done by passing a reference to itself as a parameter In the method call.

While the registry represents a single point of failure in the IGOR system, it can also be replicated for fault tolerance. The IGOR system uses a feature of CORBA to enhance the availability of the registry service. CORBA automatically can restart the registry object if it crashes. In the original IGOR system in [11], the registry maintains a database of the replica group membership in permanent storage, to recover the information in case of a crash.

Figure 4.1 illustrates the process of obtaining a reference to the registry object. The

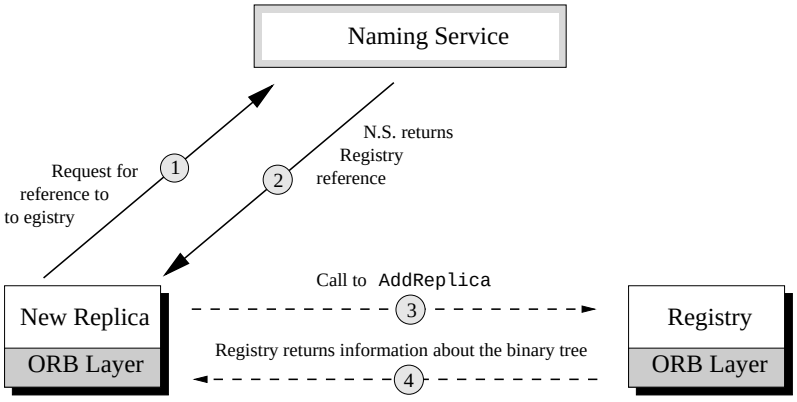

Figure 4.1: Action of the Naming Service

program only needs to make a call to bind to the registry. Since the replica is a CORBA object, it has an ORB layer that knows how to find the naming service. It then contacts the naming service, requesting a reference to the registry, by providing its interface name, `RegInterface` (indicated by the arrow labeled 1). The naming service uses the interface name to locate the registry, and responds to the new replica providing the reference to the registry object (arrow 2). Note that these two operations take place as a result of the `bind` operation. The interaction with the naming service is hidden from the user.

On the other hand, the operations indicated by arrows 3 and 4 in figure 4.1 are part of the operation of MIGOR. After receiving the reference to the registry, the new replica passes a reference to itself to the registry, for the registry to include it in the binary tree.

Upon receiving the reference of the new replica, the registry adds the new replica to the binary tree. Since the registry saves references to all the replicas, it can use them to communicate with the replicas as well. For instance, suppose the registry needs to tell the newly added replica about its neighbor replicas in the tree. The registry calls a set of methods on the replica to pass it references to the replica's parent and children. Likewise, if this replica attached itself to a certain branch in the tree, this means that one of the existing replicas now has a new child. This replica is also contacted by the registry to let it know about the new child.

Figure 4.2 illustrates the process of adding the first replica in the tree and two other replicas afterwards. Initially, the tree is empty as shown in the figure. The registry object receives a request from a new replica "A" to be registered. Since the tree is empty, the

registry places the reference to replica "A" at the root of the tree. Next, a new replica "B" requests registration as well. The registry object places the new replica's reference at the left child of the root node. Therefore, it needs to let the root of the tree know about its new child, and it also needs to let the new replica know which replica is its parent. The third replica to join the group ("C"), presents the same request to the registry. This replica gets placed as the right child of the root node. It receives information about its parent from the registry, and its parent (replica "A") receives also an update from the registry to notify it of its new right child.

During the operation of MIGOR, new replicas (servers) can join the tree at any time. The binary tree can also be rebuilt when a replica leaves the group, or when it is not responsive anymore. This feature is not a part of MIGOR currently, but it is a necessary feature to insure fault tolerance. The IGOR implementation in [11] includes this capability.

4.4 Client contacting server

A client application may contact any of the replicas of the server to retrieve the data of the current state. It is possible to contact a specified replica in particular, or a random one. The client application can specify a name along with the interface name when performing the `bind` operation, and thus bind to a specific replica. The other possibility is to indicate only the interface name, and let the ORB subsystem (the naming service) return a reference to an implementation of that interface. The client in this case has no knowledge about which replica it is communicating with. This is the usually desired mode of operation, given that the server replication process ought to be transparent to the client application.

Therefore, by simply contacting a particular type of interface, the client always sees in effect the same interface. The same data can be retrieved from the server in two consecutive queries, even if the client contacts different copies of the server. Figure 4.3 depicts the fact that the group of replicas presents a single interface to the client, and it looks like a single server to the client.

If the replica that the client has contacted crashes during a transaction, the client will simply see a failure and will try the transaction again. This time it will get attached to a different replica, and it can try the transaction once again.

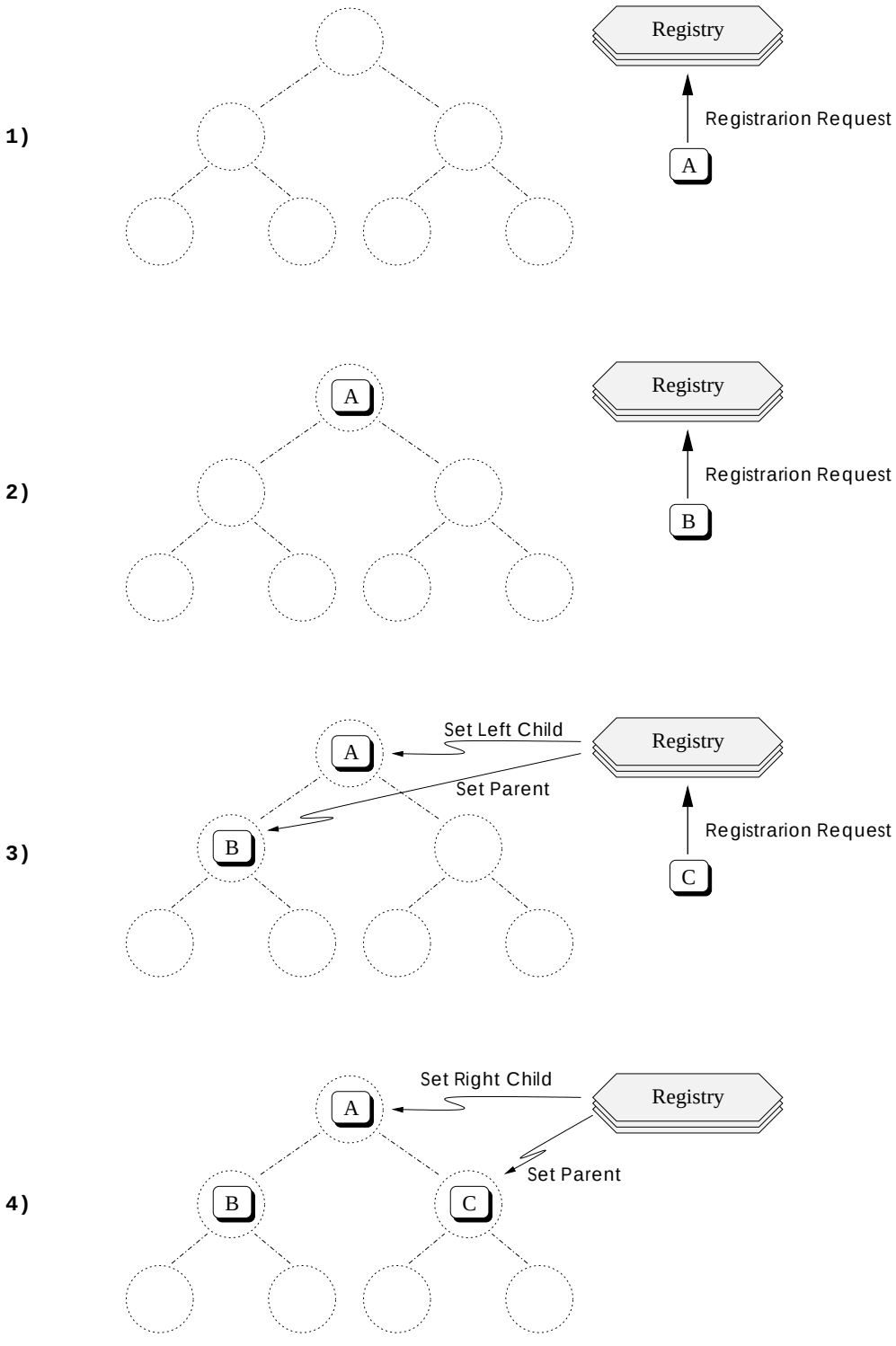

Figure 4.2: Adding Replicas to the Tree

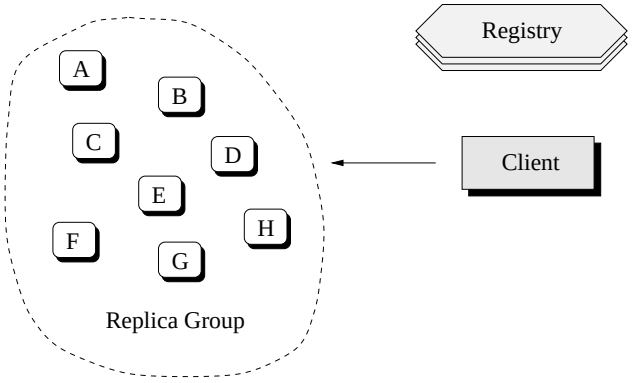

Figure 4.3: Client contacting the Server

4.5 Server State Update

There is no communication among the replicas when the client tries to simply retrieve information about the current state. However, when the client is performing an update on the data, all the replicas need to be updated so as to provide a homogeneous response to other client requests.

In this case the client contacts an individual replica as well. This replica is then responsible for propagation of the new data to the other replicas in the group. After all the replicas have received the new information, the replica that received the client request can reply to the client indicating that the operation was completed successfully. It is important to emphasize once more that the client has no knowledge about the group of replicas. To the client, the update transaction seems to take place between itself and the server it has contacted.

The method used to carry out the state change among the replicas represents the novel contribution of this thesis. IGOR [11] uses a two phase commit protocol [13], along with a binary tree arrangement to distribute messages to all replicas efficiently. This is explained in detail in section 3.2. MIGOR introduces a modification in the way messages are sent to the group. In the two phase commit protocol, the replica that receives the client request to change the state sends one message to the other replicas, collects responses from all, sends another message and collects responses once again.

MIGOR uses multicast to send messages to all the replicas. A representation of the

multicasting of messages to the group can be seen in figure 4.4. In this picture, replica "A" was the one to receive the client's request. It is therefore the one that is responsible for distribution of the information to the rest of the replicas, through a multicast. The curves in figure 4.4 indicate that there is no specific path that the messages follow from replica "A" to each of the other replicas. Instead, the message reaches all the replicas simultaneously.

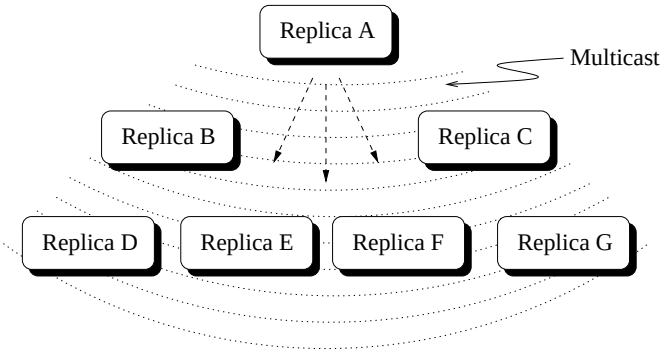

Figure 4.4: Multicast of a message

A multicast transmission is used both for the *SetState* and *Commit* messages, since both of them originate in one replica and are sent to all the others. On the other hand, the collection of responses from the group of replicas is done using the binary tree arrangement as explained in section 3.3. This method is more efficient than having the replicas multicast the responses. That is, by using the binary tree, replicas "D" and "G" in figure 4.5 can respond to their parents ("B" and "C" respectively) simultaneously. In addition to this, "B" and "C" may also be running on separate CPUs, and do not contend with each other for the CPU. This means that the responses are collected in parallel in both branches of the main tree. In addition to this, the every replica has to handle a maximum of three operations: two responses from its children, and a response to its parent. On the other hand, if one replica ("A" for instance) had to collect responses from all other replicas, it would have to handle all the responses arriving at the same time. Replica "A" would have to handle a large load of responses, and the solution would not scale as larger groups of replicas are used.

Figure 4.6 shows the state machine that governs the sequence of events that every replica goes through during the reception of a new message. The square boxes indicate the states

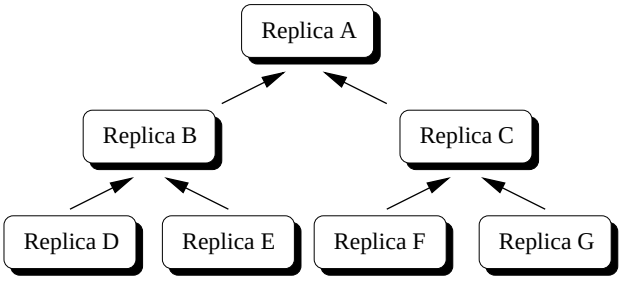

Figure 4.5: Collection of Responses

in which the replica is waiting for a multicast, at the beginning of both phases in the two phase commit protocol. The circles indicate the transition states between the other two. In the event that the transaction fails while the replica is in one of the three waiting states, a *Rollback* message will be received. The replica then aborts the operation, and returns to wait for a new *SetState* message.

An application that uses multicast will inherit the unreliability characteristic of multicast. However, in the case of MIGOR, the main source of reliable service is not the transport layer protocol used, but the two phase commit protocol that is implemented at the application layer level. In the implementation of the original IGOR the two phase commit protocol was also used to ensure reliable delivery of information in case of a failure in the IIOP communication mechanism. IIOP is essentially TCP communication.

In MIGOR, multicast is used as a replacement for TCP. While this change may mean that failures can happen at the network layer, these will be dealt with by the two phase commit protocol, yielding the same reliability level that the IGOR implementation provided.

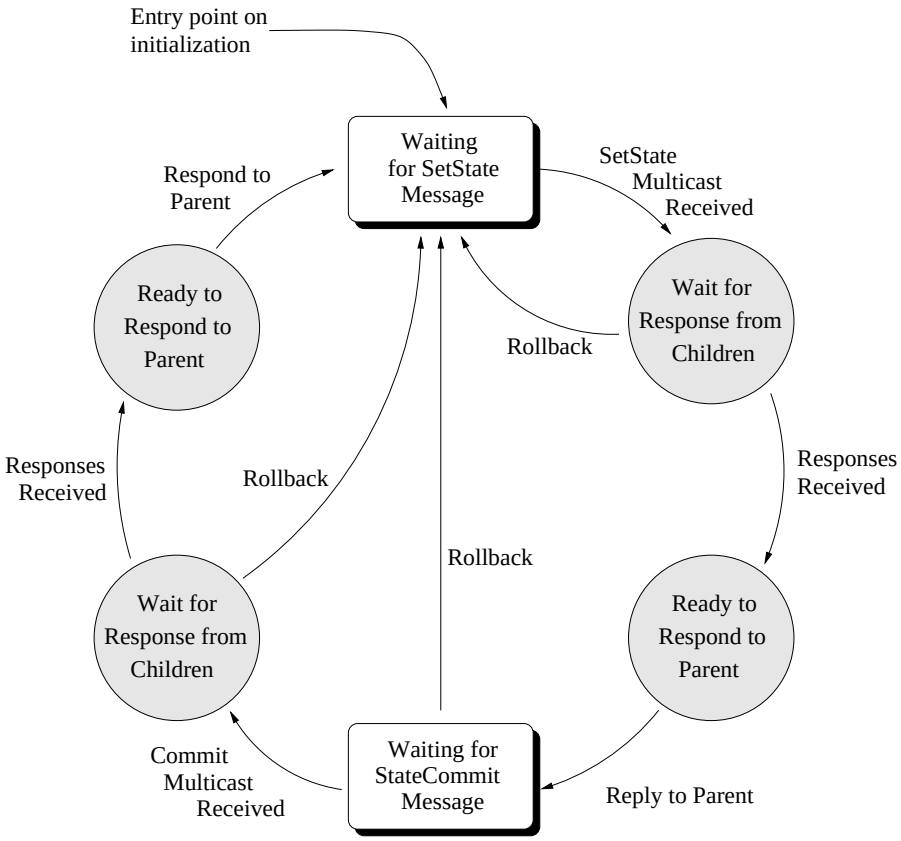

Figure 4.6: IGOR State Machine

Chapter 5

Multicast Network Programming

Programming a multicast application involves writing code at the network layer level. Since multicast works with UDP sockets, the programming of client/server applications that use multicast is very similar to the use of UDP unicast.

5.1 Sender Code

The sender of the multicast only needs to send information to a specific IP address, which is chosen to be a multicast IP address from the range 224.0.0.0 to 239.255.255.255. No other special set up is required from the server (the sender). The following is a summary of the steps necessary to set up a multicast sender using the C language's API (Application Programming Interface).

1. Open a UDP socket. `sendSock` is an `int` data type used to store the socket descriptor. The `AF_INET` tag specifies that this socket will be used for IP protocol communication, and `SOCK_DGRAM` indicates that this socket is going to be used in UDP mode [17].

   ```
   sendSock = socket(AF_INET, SOCK_DGRAM, 0);
   ```

2. Set up the destination address; i.e., the address of the multicast group. A `sockaddr_in` structure is filled in with the information about the multicast group.

   ```
   struct sockaddr_in multicast_address;
   multicast_address.sin_family       = AF_INET;
   ```

```
multicast_address.sin_addr.s_addr  = inet_addr(MCAST_GROUP);
multicast_address.sin_port         = htons(GROUP_PORT);
```

where the `MCAST_GROUP` constant denotes the IP class D address of the group to send to and `GROUP_PORT` is an `int` constant with the local port number assigned for multicast communication.

3. Send data to the multicast group. This function sends `PACKET_SIZE` bytes from the buffer `data_buffer` to the address specified by the structure `multicast_address`, through the socket `sendSock`.

```
sendto(sendSock, data_buffer, PACKET_SIZE, 0,
                 multicast_address, sizeof(multicast_address);
```

5.2 Receiver Code

The implementation of multicast in the receiver end (client) of the communication channel involves setting up the network protocol to accept IP packets destined for the multicast address. These are the steps used to set up a multicast client.

1. Open a UDP socket to receive data:

```
recvSock = socket(AF_INET, SOCK_DGRAM, 0);
```

2. Create a `sockaddr_in` structure with the information about the multicast group:

```
struct sockaddr_in multicast_client;
multicast_client.sin_family       = AF_INET;
multicast_client.sin_addr.s_addr  = htonl(INADDR_ANY);
multicast_client.sin_port         = htons(GROUP_PORT);
```

where the constant `INADDR_ANY` indicates that messages can be received from any host. The constant `GROUP_PORT` specifies the port to listen to for multicasts.

3. Bind the receiving socket with the structure that contains the multicast information:

```
bind( recvSock, (struct sockaddr *)(&multicast_client),
                sizeof(multicast_client));
```

4. Set up a structure of type `ip_mreq` with the address of the multicast group that we wish to join:

   ```
   struct ip_mreq  multicastReq;
   multicastReq.imr_multiaddr.s_addr = inet_addr(MCAST_GROUP);
   multicastReq.imr_interface.s_addr = htonl(INADDR_ANY);
   ```

   where `MCAST_GROUP` gives the class D IP address of the multicast group to join.

5. Configure the socket to listen to multicasts:

   ```
   setsockopt(  recvSock,  IPPROTO_IP,  IP_ADD_MEMBERSHIP,
                      &multicastReq, sizeof(multicastReq);
   ```

   A socket has several levels of options. This function call operates on the `IPPROTO_IP` level, to add this socket to a multicast group (through the option `IP_ADD_MEMBERSHIP`) using the multicast information included in the `multicastReq` structure.

The IP layer in this host will now accept packets destined to the address `MCAST_GROUP`, to be used by the client application.

5.3 Multicast Encapsulation

The implementation of multicast functionality is written using C functions. To facilitate the interaction with the other parts of IGOR, in MIGOR the multicast code is encapsulated within a C++ class that has a set of methods to make use of the multicast. This class, called `McastModule`, is explained in detail in section 6.3.1. This class provides the basic I/O methods needed to send and receive multicast data. Another class, `Multicast`, introduces a higher level interface. This class can also be equipped with a CORBA interface. `McastModule` provides a general low level interface to multicast, while `Multicast` provides the interface with specific functionality for MIGOR.

5.4 Multicast vs. IIOP

In this thesis all multicast communications bypass the CORBA IIOP protocol process. The advantage that multicast provides thus comes with the added cost of having to introduce into a fully CORBA compliant multicast class a great deal of functionality only found in CORBA's IIOP. The most important of these is the handling of *data marshaling* by the ORB. Using multicast to bypass the IIOP channel implies that we lose the automatic conversion of data types between different computer architectures. The current realization of MIGOR does not implement data marshaling, and hence the use of MIGOR is restricted to operation on a single type of platform.

It is certainly feasible to create a multicast-capable IIOP protocol that would add the advantage of multicast communication to the existing CORBA features. Such implementation is beyond the scope of this work.

The CORBA specification for the future version 3.0 of CORBA include means for event and message handling that may provide a more direct access to multicast, depending on the implementation.

DAIS Multicast Event Service

The Object Software Laboratories at ICL [1] have developed a multicast event service integrated into an ORB. The multicast facility is incorporated in the CORBA event service. The event service is a framework to allow *suppliers* of messages to send messages to *receivers* through an *event channel*. The DAIS Multicast event service extends this concept to have multiple receivers of a single message.

DAIS uses a form of reliable multicast based on negative acknowledgements with retransmissions from the original sender. To avoid the NACK-implosion problem, a delay is introduced in the receiver before sending a retransmission request.

Chapter 6

MIGOR Implementation Description

Incorporating multicast into IGOR involved a new, "from the ground up" implementation of IGOR in addition to developing the multicast layer for IGOR. Hence this chapter does not describe simply an addition to the previous IGOR implementation, but rather a new implementation at all levels.

6.1 IGOR Evolution into MIGOR

The incorporation of multicast into IGOR introduced a few implementation issues that led to a need for a completely new implementation of the original IGOR. The use of multicast instead of the unicast-based CORBA remote method invocation forces a few changes in the way fault tolerance is implemented. At the same time, an alternative method to accomplish class inheritance was used to simplify the design and make it more extensible.

The new IGOR, in its multicast version (MIGOR), at a fundamental architectural level, operates in the same way as the original IGOR in [11], as described in chapter 3. However, given that the IGOR architecture is not the main focus of this work, the new IGOR implementation incorporates the basic functionality, but not the rest of the functionality that can be found in the original IGOR.

This section describes the main parts of the design in which IGOR and MIGOR differ.

6.1.1 Data Type Conversion

Multicast communication was intended as a replacement for the common CORBA operation of invoking a method on a remote object, to effectively perform this method call on a set of remote objects simultaneously. In CORBA, the developer needs only to invoke a method on a local reference to the remote object, and CORBA performs the necessary network communication to perform the method call on the actual remote object, and possibly return some information. The IIOP communication system that CORBA uses is based on the TCP protocol, which provides a one-to-one (unicast), connection-oriented reliable service. This is the very part of CORBA that is being replaced when we are introducing multicast.

When we paint the picture of IGOR with multicast, MIGOR has to use multicast instead of TCP. This implies that the incorporation of multicast into IGOR must either require changes in CORBA, or use an add-on to CORBA. The second option was chosen, as it allows for modularity in the design of multicast services for CORBA. Such a module would provide a CORBA-like interface for sending data through multicast, but the actual operation of sending the data would lack the data marshaling performed by CORBA. That is, if a multicast is to be distributed among objects that are located in differing platforms, the interpretation of the data at the receiver end may vary from host to host. For example, the byte ordering of two different processors would lead to erroneous data interpretation if data is exchanged between these two processors via multicast. CORBA solves this problem by *marshaling* the data into a platform-independent format that allows for transmission as a raw stream of bytes. The ORB receiving the data has the knowledge to transform such data into the appropriate data type for the machine on which it is running.

A solution to the problem of data conversion is the CORBA *externalization service*, which is described in the CORBAservices manual, by the OMG [14]. By using this service, it is possible to convert a CORBA object, or any CORBA data structure for that matter, into a form that is suitable for transportation over the network, or for storage. This service could be used in MIGOR to make the multicast module able to perform data marshaling in the same way that a normal CORBA remote method invocation does.

The externalization service was not used in the implementation of MIGOR, however, due to the fact that it is not available in the Visibroker ORB used.

6.1.2 Client Communication

MIGOR implements a simple binary tree that allows for insertion of replicas in a tree, ordered by a label associated to each replica. The original IGOR allows also for deletion of nodes, rebuilding of the tree in case of a replica failure, and load balancing during the insertion of new replicas in the tree [11]. In this sense MIGOR is fairly limited, but the performance benefits of the multicast communication can be evaluated with this basic binary tree.

Once the tree is built, the client ought to be able to contact any of the replicas in the group to update the current state. This implies that if the replica that receives the client request is not at the root of the binary tree, it needs to propagate the new state both to its children and to its parent. In the same way, it has to collect responses from all three sides. The current implementation of MIGOR allows for the client to contact only the root of the tree in order to gather all the responses correctly.

It is possible, however, to obtain the current state of the system from any of the replicas.

6.1.3 Object Database

IGOR has an object database that maintains information about the current membership in the group of replicas. Should the registry service fail, it can be restarted immediately and read the object database to obtain the list of objects currently available. Upon doing so, it builds a new binary tree with the existing objects and the system is ready to handle replicas leaving and joining the group.

This is another part of the larger IGOR framework that was not implemented in MIGOR.

6.2 Inheritance in CORBA and C++

One of the original goals of IGOR was to add fault tolerance to an existing application without major modifications to the code in the application [11]. The original intention was that the application object would inherit fault tolerance from another object, the IGOR object. The application object would simply need to overwrite a few methods found in IGOR. The fault tolerant transaction would be handled by the IGOR object, transparently to the application.

Due to some implementation problems that arise in combining CORBA multiple inheri-

tance with C++ multiple inheritance, the actual way to incorporate IGOR into an existing application is to create an IGOR object inside the application object. The methods that could have been overwritten otherwise have to use another proxy method in the application object to access the IGOR object's method. Because of these problems, the implementation of IGOR turned into a more complex architecture than it would have otherwise been, and the impact on the application programmer was also larger than desired.

The approach taken in the new version of IGOR avoids those problems. A CORBA application must be defined through an interface definition specified in Interface Definition Language (IDL). The IDL code is used by a CORBA compiler to produce C++ code. This code implements base classes and virtual methods from which the developer has to derive the specific functionality for the application. Let us suppose that a new class is to inherit from two different IDL interfaces. This means that the IDL for the new class must inherit from the other two IDL definitions, and the C++ implementation of the new class must also inherit from the corresponding C++ implementations. Figure 6.1 shows the problematic inheritance tree. The IDL interface "C" inherits from IDLs "A" and "B". All three IDL definitions produce skeleton classes. If there are implementation objects in C++ for skeletons "A" and "B", then the implementation for "C" could inherit from them, as well as inheriting from its own skeleton class ("C").

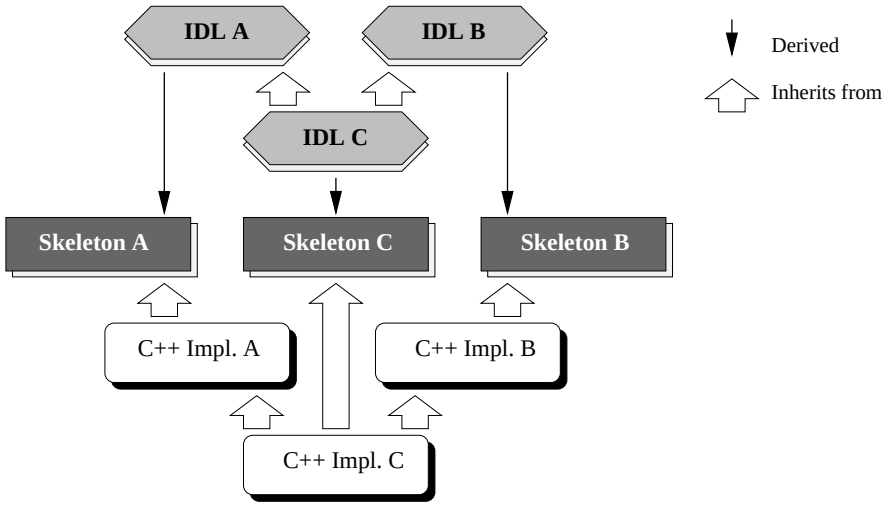

Figure 6.1: Inheritance Problem Illustration

However, this cannot be done using the Visigenic Visibroker ORB. It is not a problem of CORBA, but a problem of this particular implementation of CORBA. Visibroker generates a particular method in the skeleton of a class, that is present in all the skeletons. Therefore, the implementation of "C" will inherit this method via its own skeleton, and also via the two implementations of "A" and "B". When trying to compile this inheritance tree, the compiler cannot determine which of the three implementations of this method to use.

In the IGOR framework, an application object may have to inherit from an IGOR object and from other CORBA objects. This is the same situation as in figure 6.1. CORBA provides another mechanism to enable inheritance, that can be used to bypass this problem. This is the concept of *tie classes*. The C++ code generated by the IDL compiler includes a template definition of a class called a "tie class". It includes virtual method definitions that must be implemented in a normal C++ class. This C++ class normally would inherit from the skeleton class with the approach that one typically applies and was applied in IGOR. When using tie classes, on the other hand, the inheritance is not specified in the declaration of the C++ class implementation. Rather, a separate C++ class is created and then "tied" together with the skeleton class, through the use of the corresponding tie class, at runtime.

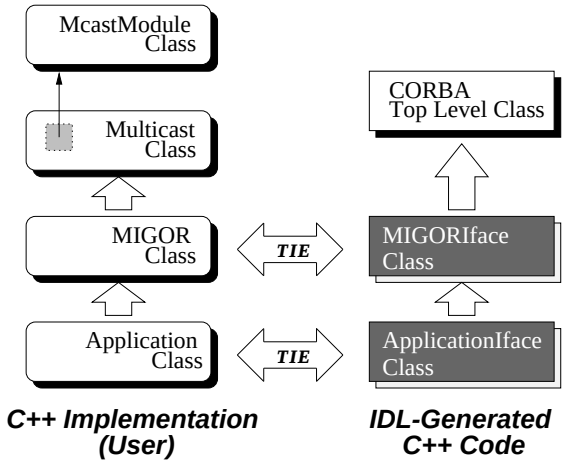

Figure 6.2: Inheritance Tree

This can be seen in figure 6.2. The white boxes denote the pure C++ implementation of the functionality that the program is to have. The dark boxes denote the C++ code resulting from the compilation of the corresponding IDL files. The upward white arrows

indicate inheritance. The box from which the arrow originates inherits from the box pointed to by the arrow.

The inheritance structure on the interface definition side must be paralleled by the same inheritance relations between the implementation objects. These objects are not limited to inheriting from the skeleton classes. Figure 6.2 shows that the classes `MIGOR` and `McastModule` are not tied to skeleton classes. This is because these classes do not have a interface definition in IDL. The class `MIGOR` does not inherit from `McastModule`, but rather contains an instance of that class.

6.2.1 TIE Class Approach Example

A comparison of the normal way of CORBA server programming and the tie class approach is presented in this section. The first section of code contains the definition of all the IDL interfaces involved. These definitions are the same for both approaches.

IDL Objects Definitions [COMMON] **Objects.idl**

```
 1:  interface aclass {
 2:   void method_a();
 3:   };
 4:
 5:  interface bclass {
 6:   void method_b();
 7:   };
 8:
 9:  interface cclass : aclass, bclass {
10:   void method_c();
11:   cclass test();
12:   };
```

The interface definitions for classes `aclass` and `bclass` are at lines `1` and `5`. Each interface defines only one method. The interface `cclass` inherits from the other two (line `9`). The syntax of IDL is fairly similar to C++.

Usual Inheritance Approach

In the next section of code one can find the C++ header with the definitions of the implementation of the interfaces first defined in the IDL:

C++ Objects Definitions [NON-TIE] **Objects.h**

```
 1:  // **
 2:  // ** Class  AClass
 3:  // **
 4:  class AClass : public _sk_aclass {
 5:   long  _class_data;
 6:  public:
 7:   AClass( void );
 8:   AClass( int class_data );
 9:   virtual ~AClass( void );
10:   virtual void method_a( void );
11:   };
12:
13:  // **
14:  // ** Class  BClass
15:  // **
16:  class BClass : public _sk_bclass {
17:   long  _class_data;
18:  public:
19:   BClass( void );
20:   BClass( int class_data );
21:   virtual ~BClass( void );
22:   virtual void method_b( void );
23:   };
24:
25:  // **
26:  // ** Class  CClass
27:  // **
28:  class CClass : public AClass, public BClass {
29:   long  _class_data;
30:  public:
31:   CClass( void );
32:   CClass( int class_data );
33:   virtual ~CClass( void );
34:   virtual void method_c( void );
35:   };
```

In the file `Objects.h` we can find the definitions for classes `AClass` and `BClass` at lines `4` and `16`, respectively. Note that these two classes inherit from `_sk_aclass` and `_sk_bclass`, which are the skeleton classes generated by the IDL compiler from the IDL code. As seen at line `28`, the class `CClass` inherits both from `AClass` and `BClass`.

The next step is to implement the methods in C++. The code for doing so is the same for both the non-tie class and the tie class approaches. All the method implementations can be found in this code listing.

C++ Implementations [COMMON] **Objects.C**

```
 1:  #include <Objects.h>
 2:
 3:  // **
 4:  // **  METHOD DEFINITIONS: AClass
 5:  // **
 6:  AClass::AClass( void ) {
 7:   _class_data = 0;
 8:   }
 9:
10:  AClass::AClass( int class_data) {
11:   _class_data = class_data;
12:   }
13:
14:  AClass::~AClass( void ) {
15:   _class_data = 0;
16:   }
17:
18:  void AClass::method_a( void ) {
19:   cout << "AClass::method_a() "                << endl;
20:   cout << "  _class_data is : " << _class_data << endl;
21:   }
22:
23:  // **
24:  // **  METHOD DEFINITIONS: BClass
25:  // **
26:  BClass::BClass( void ) {
27:   _class_data = 0;
28:   }
29:
30:  BClass::BClass( int class_data) {
31:   _class_data = class_data;
32:   }
33:
34:  BClass::~BClass( void ) {
35:   _class_data = 0;
36:   }
37:
38:  void BClass::method_b( void )  {
39:   cout << "BClass::method_b() "                << endl;
40:   cout << "  _class_data is : " << _class_data << endl;
41:   }
42:
43:  // **
44:  // **  METHOD DEFINITIONS: CClass
45:  // **
46:  CClass::CClass(void) : AClass(0), BClass(0) {
47:   _class_data = 0;
48:   }
49:
```

```
50:  CClass::CClass(int class_data): AClass(class_data), BClass(class_data) {
51:   _class_data = class_data;
52:   }
53:
54:  CClass::~CClass( void ) {
55:   _class_data = 0;
56:   }
57:
58:  void CClass::method_c( void ) {
59:   cout << "CClass::method_c() "                << endl;
60:   cout << "  _class_data is : " << _class_data << endl;
61:   }
```

As a result of this implementation, the `CClass` is now a class with its own method (`method_c`) and with two inherited methods, `method_a` and `method_b`. Since classes `AClass` and `BClass` inherit from `_sk_aclass` and `_sk_bclass`, the link is established between the C++ implementation and the CORBA interface for the three methods mentioned above. This is what is done differently when the tie class approach is used, as we will see in the next example.

Finally, the main function in the server program creates a CClass object at line 7 in the following listing.

Server Program [NON-TIE] **Server.C**

```
 1:  #include <Objects.h>
 2:
 3:  int main( int argc, char **argv, char ** ) {
 4:   CORBA::ORB_var  orb = CORBA::ORB_init( argc, argv );
 5:   CORBA::BOA_var  boa = orb->BOA_init( argc, argv );
 6:
 7:   CClass               c_obj( 1024 );
 8:
 9:   boa->obj_is_ready( &c_obj );
10:   boa->impl_is_ready();
11:   }
```

A reference to the newly created object (`c_obj`) is passed to the Basic Object Adapter (BOA) at line 9.

Tie-Class Approach

Two of the files explained above (`Objects.C` and `Objects.idl`) do not present any changes when using the tie class approach to do inheritance. However, the C++ definition

of the implementation of the classes does need some changes to use the tie class approach. The following code listing contains these changes:

C++ Objects Definitions [TIE] **Objects.h**

```
 1:  // **
 2:  // ** Class  AClass
 3:  // **
 4:  class AClass {
 5:   long  _class_data;
 6:  public:
 7:   AClass( void );
 8:   AClass( int class_data );
 9:   virtual ~AClass( void );
10:   virtual void method_a( void );
11:   };
12:
13:  // **
14:  // ** Class  BClass
15:  // **
16:  class BClass {
17:   long  _class_data;
18:  public:
19:   BClass( void );
20:   BClass( int class_data );
21:   virtual ~BClass( void );
22:   virtual void method_b( void );
23:   };
24:
25:  // **
26:  // ** Class  CClass
27:  // **
28:  class CClass : public AClass, public BClass {
29:   long  _class_data;
30:  public:
31:   CClass( void );
32:   CClass( int class_data );
33:   virtual ~CClass( void );
34:   virtual void method_c( void );
35:   };
```

The differences with respect to the file `Objects.h` used in the non-tie class approach can be seen at lines `4` and `16`. The new definitions of the first two classes (`AClass` and `BClass`) do not inherit from the respective skeleton classes, like in the previous case. Since there is no link here between the IDL interface (skeleton classes) and the implementation in C++, the link must be done somewhere else. When using tie classes, this linking is done in the `Server.C` file:

Server Program [TIE] **Server.C**

```
 1:  #include <Objects.h>
 2:
 3:  int main( int argc, char **argv, char ** ) {
 4:   CORBA::ORB_var  orb = CORBA::ORB_init( argc, argv );
 5:   CORBA::BOA_var  boa = orb->BOA_init( argc, argv );
 6:
 7:   CClass               c_obj( 1024 );
 8:   _tie_cclass<CClass>  c_tie( c_obj );
 9:
10:   boa->obj_is_ready( &c_tie );
11:   boa->impl_is_ready();
12:   }
```

In this code listing, at line 7, an object of type `CClass` is create, like in the previous case. However, this is not the object whose reference is passed to the BOA. Instead, a new object (`c_tie`) is created, of type `_tie_cclass<CClass>`. This data type is a *tie class*, implemented as a template. The type `CClass` is used in the template `_tie_cclass` to link the implementation (`CClass`) to the IDL interface definition (`cclass`). The object of this type is the one that is finally passed to the BOA to activate the implementation.

6.3 Implementation Details

A description of the IDL interfaces and C++ implementation of the different parts of MIGOR is presented in this section.

6.3.1 McastModule Class

The constructor for this class sets up the IP layer to use multicast, as introduced in section 5. The following section of code shows the definitions of the methods in this class.

McastModule Method Definitions

```
102:    int             sendData(char *, int);
103:    ObjectState *   recvData(struct sockaddr *);
104:    ObjectState *   recvData();
105:    boolean         readyTo(actionType);
106:    int             setClientAddr(struct sockaddr *);
107:    int             writen(int, const void *, int);
108:    int             readn(int, const void *, int);
109:    char            getReceivedType();
```

There are two methods that give access to multicast communication (at lines 102 and 104). These are `sendData` and `recvData`. The first one takes two arguments: a pointer to a buffer with the data to send, along with an `int` value specifying the number of bytes to be sent. There are two versions of the method for receiving a multicast. At line 103 there is a definition of a method that takes as an argument a `sockaddr` structure pointer. The second method definition does not take any arguments, but both versions return a pointer to a `ObjectState` object. When a multicast is received, it is possible to retrieve the address that originated the multicast. For this reason, the `recvData` method can also return this information to the user through a `sockaddr` structure. If the user is not interested in this information, the other definition can be used, which simply makes a call to the first definition of the function, discarding the return address information. In either case the method returns a pointer to an ObjectState object, with the new state just received through the multicast.

Method `readyTo` at line 105 is meant to allow the user to probe the multicast socket to determine whether it is ready to read or write data. The last method of interest to the user of this class is at line 109. `getReceivedType` returns the type of data received: either new data (identified by `D`) or a commit message (`C`). The rest of the methods are used internally by `McastModule`.

6.3.2 Multicast Class

This class is designed to be a wrapper class around the `McastModule` class. It has similar methods to `McastModule`, which form a specific data packet for either a SetState message or a Commit message, for instance.

Multicast Class Definition

```
115:  class Multicast {
116:  private:
117:    ObjectState laststate;
118:  public:
119:    McastModule *   mcastc;
120:    McastModule *   mcasts;
121:    int             seq_number;
122:
123:    Multicast(const char *object_name=NULL);
124:    virtual ~Multicast();
125:    char            getData();
126:    CORBA::Long     setData(const ObjectState& newstate,
```

```
127:                                                CORBA::Long new_size);
128:    CORBA::Long     setCommit();
129:    virtual void    McastReceived() = 0;
130:    ObjectState     getState();
131:    };
```

This class contains two instances of `McastModule` objects; one for sending multicasts and another one for receiving. That is, an application that inherits from the `Multicast` class can act both as a multicast server (sender) and client (receiver). This way it is potentially possible to send and receive on different multicast groups.

At line `126` there is the definition for `setData`, which is used to send the state contained in `newstate`, of size `new_size` bytes. It is worth noting that the arguments as well as the return values are defined using CORBA data types. This is so because this class could also be implemented as a CORBA object, to facilitate the integration of this class in a CORBA system. This would be more relevant when an externalization service can be used together with this class to encapsulate the data into a platform-independent form. That would be the most suitable way to use this combination of the `Multicast` class with the `McastModule` object.

However, the current implementation of MIGOR does not include a CORBA version of the `Multicast` class, because of the lack of an externalization service.

Similarly, the method `setCommit` at line `128` works like `setData`. It prepares a data packet with a commit message to send it by way of the methods in the `McastModule` object to the multicast group. The method `getData` is used to retrieve the data that has arrived at the multicast socket, be it a SetState message or a Commit message. The method `getState` returns the actual state after being retrieved by `getData`.

The `McastReceived` method is defined as pure virtual, to force it to be overwritten by any class that derives from this one. Specifically, the `MIGOR` class will implement this method. When a multicast is received, this method is called *in the class* `Multicast`, but the `MIGOR` class is the one to take the appropriate action after the multicast has been received. For this reason, `MIGOR` must overwrite this method, so that its method gets called upon the reception of a multicast. Please note that we have not specified from where the call to `McastReceived` comes. This is explained in the next section.

6.3.3 Dispatcher Class

The `Dispatcher` class contains methods to handle specific events. It implements one method only, to handle the arrival of a multicast packet at the network interface. This class is used in conjunction with a signal handler.

There are two approaches to finding out when a multicast has arrived at a UDP socket: polling the socket constantly, or catch a special signal raised by the operating system when new data has arrived at a socket. The first solution makes intense use of the CPU, and is not appropriate. The solution for this project was to catch this special signal, called `SIGIO`, through a signal handler.

The signal handler is precisely the method implemented by this class. `IncomingMcast` gets called when a signal `SIGIO` is generated by the O.S. [18]. This method, in turn, is the one to call `McastReceived` in the object `mcast` of type `Multicast`. This reference (`mcast`) is obtained when the `Dispatcher` object is created. This occurs during the initialization of the `MIGOR` object, whose explanation follows in the next section. `MIGOR` inherits from `Multicast`, so `MIGOR` can pass a reference to itself to the `Dispatcher` object when `MIGOR` creates it.

Dispatcher Class Definition

```
141:  class Dispatcher {
142:  private:
143:    static Multicast *mcast;
144:  public:
145:    Dispatcher() { mcast = NULL; }
146:    Dispatcher(Multicast *);
147:    int IncomingMcast(int);
148:    };
```

The signal handler for the signal `SIGIO` is a method (`IncomingMcast`) in an object (`Dispatcher`) that is created dynamically inside the `MIGOR` object. In addition to this, the reference to `Multicast` inside the `Dispatcher` is a pointer. Because of all of this, this pointer to `Multicast` does not keep its value at run time, when the signal occurs. For this reason, the reference `mcast` has to be defined static, as seen at line `143`, to ensure that the signal will trigger the signal handler.

6.3.4 MIGOR IDL Interface

The MIGOR class contains the transaction processing functionality to coordinate the state updates among all the replicas. This class is the one that makes decisions based on the multicast packets received, or the responses received through the tree. This object communicates with the other MIGOR objects in the other replicas. The responses in the two phase commit protocol that are passed up traversing the tree are CORBA calls. The methods used in this communication are defined in the MIGOR IDL file:

MIGOR IDL Definition

```
 1:  #include "McastInterface.idl"
 2:
 3:  interface MigorInterface {
 4:   long SetState(in ObjectState newstate, in long size);
 5:   long Commit();
 6:   long setParent(in MigorInterface parent_ );
 7:   long setLchild(in MigorInterface lchild_ );
 8:   long setRchild(in MigorInterface rchild_ );
 9:   long num();
10:
11:   // ** TP functionality:
12:   void StateResponse();
13:   void CommitResponse();
14:   };
```

The two methods at lines 13 and 14 are the ones used by the children to perform the responses in both phases of the two phase commit protocol. At lines 4 and 5 are the function definitions for the SetState and Commit calls. These definitions describe the parameters they require. Even though they will be implemented by CORBA methods in the MIGOR class, these methods in turn use the multicast mechanism to distribute these messages among all the replicas.

Other necessary methods in the MIGOR class are the ones in lines 6 to 8. These are used by the registry to inform this replica about its new parent or children.

6.3.5 MIGOR Class

This class includes the same methods described earlier, since it is the implementation in C++ of the interface defined with the IDL in the previous section. These methods are defined in lines 176 through 180, and lines 187 and 188. In addition, there are three methods that this implementation adds to the ones defined by the interface. Not being

CORBA methods, they cannot be accessed remotely. They are only meant to be used by the classes that may be derived from this class.

MIGOR Class Definition

```
176:    CORBA::Long SetState(const ObjectState&, CORBA::Long);
177:    CORBA::Long Commit();
178:    CORBA::Long setParent(MigorInterface_ptr);
179:    CORBA::Long setLchild(MigorInterface_ptr);
180:    CORBA::Long setRchild(MigorInterface_ptr);
181:    CORBA::Long num();
182:
183:
184:    // ** TP funcionality:
185:    virtual void McastReceived();
186:
187:    void StateResponse();
188:    void CommitResponse();
189:    virtual void SaveNewState() = 0;
190:    virtual void CommitState()  = 0;
```

`SaveNewState` and `CommitState` are two pure virtual methods that need to be implemented by classes deriving from this one. The class `Application` is the one that has `MIGOR` as its parent class. The application maintains the current state of the application. Therefore, MIGOR only needs to make a function call on the application to indicate when to operate on the state. The `MIGOR` class calls its own `SaveNewState` and `CommitState` methods, which will in turn cause the same methods to be called in the application, since it is overwriting these two methods.

The other method included in this class is `McastReceived`, which gets called by the signal handler mechanism when a multicast is received at the socket. When this method is called, it uses the methods from the `Multicast` class (which is this object's parent class) to retrieve the data from the multicast socket. If the multicast received was a SetState message, the replica will wait for the responses from its children, if it has any. If this is not the case, or when all the responses are received, this replica will respond to its parent by calling the StateResponse method of the parent. Recall that when a replica registers with the registry, it receives information about its parent and children. This is how it obtains a reference to the parent replica, to be able to perform this StateResponse call at this point.

A similar process takes place when the received multicast is a Commit message. The replica once again waits for the responses for as many children as it has, and then responds

with a CommitResponse to its parent by making a method call on it, in the same way the StateResponse is made.

6.3.6 Server Application IDL Interface

In the application interface definition it can be seen how this interface inherits from the `MIGOR` interface (line `16`). In lines `17` and `18`, two methods are defined for setting the state and retrieving it, respectively. These methods are called by the client application.

Application IDL Definition

```
 1:  #include "McastInterface.idl"

16:  interface ApplicationInterface : MigorInterface {
17:   long NewState(in ObjectState newstate);
18:   void GetState(out ObjectState state);
19:    };
```

6.3.7 Server Application Class

This definition shows the inheritance from the `Migor` class, to mirror the inheritance relation defined in the IDL interface definition. In lines `229` and `230` one finds the definitions of the functions in the `Migor` class that have to be overwritten here.

Application Class Definition

```
218:  class Application : public Migor {
219:  private:
220:    ObjectState temp;
221:    ObjectState mystate;
222:  public:
223:    Application(const char *ob_name,long num);
224:    virtual ~Application();
225:    CORBA::Long     NewState(const ObjectState&);
226:    void            GetState(ObjectState&);
227:
228:    // * TP specific functionality:
229:          virtual void SaveNewState();
230:          virtual void CommitState();
231:    };
```

In lines `225` and `226` are the methods defined in the IDL for this class. It is also worth noting that the Application class maintains two `ObjectState` objects: one with the actual state and another one (`temp`) to store the new state during the first phase of the two phase commit protocol.

6.3.8 Registry IDL Interface

The registry is in charge of maintaining a list, or rather, a binary tree, of replicas that currently exist. Therefore, its interface definition includes a method for adding replicas. This is the only operation that the registry needs to supply for another object.

Registry IDL Interface

```
1:  #include "Migor.idl"
2:
3:  interface RegInterface {
4:   oneway void AddReplica(in MigorInterface a);
5:   };
```

This method is defined as `oneway`, which means that a call on this method does not block waiting for it to return. For instance, the server application (the replicas) obtain a reference to the registry object from the naming service of the ORB. They then invoke the method `AddReplica` on the registry reference. Because it is a oneway method, the method invocation will return immediately, without waiting for the registry to finish processing the method.

This is necessary because of the sequence of events that take place for the replica to register. This is explained later.

6.3.9 Registry Class

After compiling the IDL file with an IDL compiler, the class `_sk_RegInterface` is included in the C++ code that results. This is the skeleton class that links CORBA with the user implementation. Therefore, the `Registry` class inherits from `_sk_RegInterface`, and it has to overwrite the method on line `4` of the IDL code, to provide the implementation of `AddReplica`.

Registry Class Definition

```
199:  class Registry : public _sk_RegInterface {
200:  private:
201:    BinTree<MigorInterface> *tree;
202:  public:
203:    MigorInterface_ptr              temp1,temp2;
204:    Registry();
205:    virtual ~Registry();
206:    Registry(const char *ob_name);
207:    void AddReplica(MigorInterface_ptr a);
208:    };
```

It also worth noting that the `Registry` class contains a binary tree object, specified by the `BinTree` template. This template describes a basic binary tree with capability to insert nodes and retrieve information from the tree. The object declaration at line `201` creates a pointer (`tree`) to a binary tree of `MigorInterface` data types. That is, the tree is to hold references to `MigorInterface` types. This is how the list of replicas is kept in the tree.

6.3.10 Server Application (Replicas)

When a replica is started, it needs to communicate with the registry to add itself to the list of replicas, and to obtain knowledge about its own position within the binary tree.

Replica Code

```
23:   // Bind to the registry:
24:   reg = RegInterface::_bind();
25:   RegInterface::_duplicate(reg);
26:
27:   // Create application object
28:   Application *    this_app = new Application(name,number);
29:   _tie_ApplicationInterface<Application>app_tie( *this_app, name );
30:
31:   boa->obj_is_ready( &app_tie );
32:   reg->AddReplica( MigorInterface::_narrow( &app_tie ) );
33:   boa->impl_is_ready();
34:   }
```

In line `24` in the code listing above we can see the act of getting a reference to the registry in which a reference to an interface of type `RegInterface` is requested from the ORB. This way, the replica can find the registry automatically when it starts. The next task is to create an instance of the actual server object (of type `Application`, which is done at line `28`.

Line `29` contains the invocation necessary for realizing the "tie class" approach described earlier. An object of type `_tie_ApplicationInterface<Application>` is created with the name `app_tie`. `_tie_ApplicationInterface` is a template generated as a result of IDL compilation of the Application IDL definition. The object type passed to the template (`Application`) is the name of the class that realizes the implementation in C++ of the methods defined through the IDL definition. Therefore, through this declaration, `app_tie` will be a CORBA object using the interface of `ApplicationInterface` and the implementation of `Application`.

Chapter 7

Performance Analysis and Comparison

7.1 Analysis Environment

Given that the current implementation of MIGOR can only operate among homogeneous platforms due to the lack of an externalization service, the machines available to perform the timing analysis were four DEC Alphas running Digital Unix. The names of these hosts are `vision1`, `vision2`, `vision3`, and `fusion`. These workstations are located in the WPI Machine Vision Laboratory (MVL), and are connected through a 10 Mbps Ethernet network. See table A.1 for a description of the workstations' specifications.

The measurement of interest is time elapsed from the time that a client contacts the a replica to update the state, until the time that the replica responds to the client, after completing the update of the state in the replica group. Each measurement consists of 1000 consecutive state updates on the replica group, to record the minimum, maximum and average times. The state being updated consists of one CORBA `Long` data type. The client application is run from `xfactor`, because the SGI architecture has better clock resolution.

In order to perform measurements on a configuration with a balanced binary tree, the tests were carried out in three scenarios: using one replica, using three replicas and using seven replicas.

7.1.1 One Replica

With a single server object no intra-group communication takes place, and therefore the latency in this test equals the latency of a simple CORBA call (from the client to the server).

	Object State: 1 `Long` 1 MIGOR Replica			
Machine	Minimum	Average	Maximum	Standard Deviation
fusion	2.90 ms	3.39 ms	11.73 ms	0.57ms
vision1	4.62 ms	5.43 ms	27.81 ms	0.36ms
vision2	4.58 ms	5.93 ms	82.54 ms	3.97 ms
vision3	4.45 ms	6.00ms	120.49ms	4.96ms

Table 7.1: Latency Measurements with One MIGOR Server

In this test, and in all the rest, it is important to consider the minimum observed latency, and not the other values. The machines used for the test had other processes running, therefore, the all timings recorded in these tests represent cases in which the program running the test had to share the CPU with other processes. For this reason the maximum and average delay values do not represent the maximum achievable performance that would be obtained in a "clean" system with the only latency being that imposed by CORBA and communications processing. The minimum latency value comes closest to representing this information.

7.1.2 Three Replicas

For this test, each of the replicas runs on a separate machine. This way, when the multicast is received at the same time by all the replicas, all three of them can process it simultaneously since they are using separate processors. The replica at the root of the tree was running on `vision1`, the right child on `vision2` and the left child on `vision3`.

Object State: 1 `Long` 3 MIGOR Replicas			
Minimum	Average	Maximum	Standard Deviation
8.76 ms	13.455 ms	175.57 ms	14.71 ms

Table 7.2: Latency Measurements with Three MIGOR Servers

Considering one CORBA call between vision1 and fusion takes about 4 ms, the 8.76 milliseconds observed with three replicas corresponds to two CORBA calls which are the `SetStateResponse` and the `CommitResponse` that come from the children to the parent. The two children reply in parallel, so the two `SetStateResponse` calls come in the parent at the same time, and take only about 4 ms. The `CommitResponse` from the two children takes another 4 ms, which adds up to the approximately 8 ms observed.

7.1.3 Seven Replicas

Two variants of this test were performed. In one of them, the seven replicas were distributed among the four machines available in a way so as to avoid two of them executing on the same processor at the same time. The second set up involved running all the replicas on the same host (`fusion`) to study the impact of a sharing of the processor by a large group of replicas.

Shared Load

The following figure shows the names of the machines on which the replica in the corresponding tree location was run.

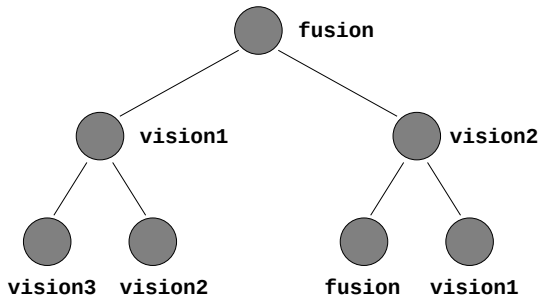

Figure 7.1: Computers Hosting the 7 Replicas

All load on `fusion`

When the test is performed with all the replicas running on the same host, we can observe that the times are approximately equal to those obtained for the shared load across machines.

Object State: 1 `Long` 7 MIGOR Replicas			
Minimum	Average	Maximum	Standard Deviation
21.45 ms	25.143 ms	150.46 ms	6.47 ms

Table 7.3: Latency Measurements with Seven MIGOR Servers on four hosts

Object State: 1 `Long` 7 MIGOR Replicas			
Minimum	Average	Maximum	Standard Deviation
20.47 ms	23.43 ms	180.23 ms	3.60 ms

Table 7.4: Latency Measurements with Seven MIGOR Servers on `fusion`

All load on `vision1`

For comparison, the test was run on vision1 as well, since fusion has two CPUs, and vision1 has one. ence the fusion based results mask some of the processor load effects that are clearly visible below.

Object State: 1 `Long` 7 MIGOR Replicas			
Minimum	Average	Maximum	Standard Deviation
53.22	61.93	398.58	13.76

Table 7.5: Latency Measurements with Seven MIGOR Servers on `vision1`

7.2 Performance Comparison

The performance of MIGOR was compared to the original IGOR system by performing the same tests with both systems. In addition to testing these two systems, a third one was also evaluated. As described in chapter 6, the contribution of this thesis was not only to add multicast to IGOR, but also to build a completely new implementation of IGOR, using a different programming approach. The timing results for 1, 3 and 7 replicas are presented in tables 7.6, 7.7 and 7.8, respectively, for all three versions of IGOR.

The test with seven replicas was performed with the load being shared among the four Digital Unix machines available, since the original timing analysis of IGOR was done this

	Object State: 1 `Long` 1 Object Replica			
System	Minimum	Average	Maximum	Standard Deviation
IGOR	1.87 ms	2.02 ms	4.74 ms	- ms
New IGOR	3.15 ms	3.85 ms	23.87 ms	0.98 ms
MIGOR	4.62 ms	5.43 ms	27.81 ms	0.36ms

Table 7.6: Performance comparison for all types of IGOR with 1 replica

	Object State: 1 `Long` 3 Object Replicas			
System	Minimum	Average	Maximum	Standard Deviation
IGOR	310.24 ms	407.41 ms	1181.51 ms	- ms
New IGOR	37.27 ms	41.82 ms	195.53 ms	7.27 ms
MIGOR	8.76 ms	13.455 ms	175.57 ms	14.71 ms

Table 7.7: Performance comparison for all types of IGOR with 3 replicas

way [11].

	Object State: 1 `Long` 7 Object Replicas			
System	Minimum	Average	Maximum	Standard Deviation
IGOR	650.64 ms	710.35 ms	1559.50 ms	- ms
New IGOR	126.41 ms	140.67 ms	640.95 ms	34.21 ms
MIGOR	21.45 ms	25.143 ms	150.46 ms	6.47 ms

Table 7.8: Performance comparison for all types of IGOR with 7 replicas

It can be seen that the performance difference between the original IGOR system and the new multicast IGOR is very large. The main reason for such difference is not a result of the use of multicast. As seen earlier, multicast is used to send two messages to the group of replicas, and the binary tree is used for collecting two other messages progressing up the tree. We might expect a non-multicast version of IGOR to show a latency of approximately twice that of MIGOR.

However, this is not the case. The reason for the big performance difference between the original IGOR and MIGOR is that the two implementations of IGOR used different design approaches. The largest factor that affects the performance of the original version of IGOR is described in a paper by Hazzard and Cyganski [8]. Some CORBA calls used in

IGOR are one-way calls. That is, they do not return any information, nor do they return a `void` type. The object that originates the call invokes the method on the object being called, and does not wait for the call to finish processing at the object called. The TCP packet used to make the call must be acknowledged, but because there is no information returned immediately, the TCP ACK cannot be attached to any returning TCP packet. Therefore, the TCP acknowledgement mechanism times out after a long period of time (as much as 200 ms) and sends a packet containing only the acknowledgement. This is the delayed ACK mechanism of TCP, which due it its interaction with the Nagle algorithm, introduces a considerable delay in implementations that use CORBA one way calls.

The greater than two-to-one performance difference between the new IGOR and MIGOR is driven by the fact that messages ascend the binary tree more quickly than they descend owing to the parallel responses that children generate versus the serial method calls that parents make on children.

Chapter 8

Conclusions

Distributed low latency systems require efficient communication among all members of the system.

The results show that a multicast transmission can certainly deliver messages to a group of receivers faster than a unicast for each receiver. In the context of distributed computing, it is clear that multicast is a very promising solution for the distribution of messages when the same message must reach a group of hosts.

However, by using IP multicast, an application inherits the unreliability of the UDP protocol used with IP multicast. There are several new protocols that provide reliable multicast at different levels. An implementation of IGOR using reliable multicast could be realized which would take advantage of an efficient multicast message delivery mechanism with TCP-like reliability.

In the framework of IGOR reliability is achieved with the two phase commit protocol. For this reason, it is advantageous to exclude the reliability service from the network layer because of the performance gain through the use of IP multicast. A distributed system like IGOR can benefit specially from the multicast transmission of messages, because in IGOR it is necessary to send the same message to many receivers.

The MIGOR system achieves fault tolerance through server object replication. The definition of fault tolerance is given in the context of transaction services. The MIGOR system is fault tolerant in that it can withstand specified faults in the services on which it depends.

8.1 Future Work

As stated in the thesis presentation, it will be necessary to use a CORBA externalization service to incorporate data marshaling capabilities in MIGOR. In order to obtain a system with comparable fault tolerance to the original version of IGOR, it will also be necessary to implement some of the features that IGOR introduced that are not present in MIGOR at the moment. Some such features are an object database, and the ability to restructure the binary tree in case one or more replicas fail.

Another potentially interesting area of research is the investigation of using a decentralized non-blocking atomic commitment protocol [7]. Such protocols allow any member of a group of objects to send a message to the group, while each host arrives individually at the decision as to whether to commit the new state or not; but all objects arrive at the same decision. The protocol operates by sending multiple messages from all objects to all other objects.

Appendix A

Machine Vision Laboratory Workstations

A.1 Hardware/Software Environment

Name	Processor	CPU Speed	RAM	Operating System
fusion	DEC Alpha	190† MHz	576 MB	Digital Unix v3.2
vision1	DEC Alpha	150 MHz	48 MB	Digital Unix v3.2
vision2	DEC Alpha	150 MHz	48 MB	Digital Unix v3.2
vision3	DEC Alpha	150 MHz	64 MB	Digital Unix v3.2
xfactor	MIPS	200 MHz	384 MB	SGI Irix 6.2

† two 190 MHz processors

Table A.1: Machine Vision Laboratory Workstations

Bibliography

[1] *DAIS Multicast Event Service*. http://www.daisorb.com/, 1998.

[2] Becker, D. Beowulf Project. Web Page, http://cesdis.gsfc.nasa.gov/linux-web/beowulf/beowulf.html. Accessed April, 1998.

[3] Coulouris, G., Dollimore, J., and Kindberg, T. *Distributed Systems Concepts and Design*, second ed. Addison-Wesley, Englewood Cliffs, New Jersey 07632, 1994.

[4] Deering, S. RFC 1112: Host extensions for IP multicasting. Tech. rep., Internet Engineering Task Force, 1989.

[5] Deering, S. E., and Cheriton, D. R. RFC 966: Host groups: A multicast extension to the internet protocol. Tech. rep., Internet Engineering Task Force, 1985.

[6] Floyd, S., Jacobson, V., McCanne, S., Liu, C., and Zhang, L. A reliable multicast framework for light-weight sessions and application framing. In *ACM SIGCOMM'95* (1995), ACM.

[7] Guerraoui, R., and Schiper, A. The decentralized non-blocking atomic commitment protocol. *IEEE International Symposium on Parallel and Distributed Processing* (1995).

[8] Hazzard, B., and Cyganski, D. Large anomalous TCP and CORBA latencies: Observation, analysis and mitigation. To be submitted for publication, 1998.

[9] Lampson, B. W., Paul, M., and Siegert, H. J., Eds. *Lecture Notes in Computer Science. Distributed Systems - Architecture and Implementation*, first ed. Springer-Verlag, 1981.

[10] Lin, J. C., and Paul, S. RMTP: A reliable multicast transport protocol. vol. 3, pp. 1414–1424.

[11] Modzelewski, B. E. Interactive-group object-replication: An approach to fault tolerance in a CORBA distributed computing environment. Master's thesis, Worcester Polytechnic Institute, 1997.

[12] Modzelewski, B. E., Cyganski, D., and Underwood, M. V. Interactive-group object-replication fault tolerance for corba. In *The Third Conference on Object-Oriented Technologies and Systems (COOTS) Proceedings* (1997), The USENIX Association.

[13] Mullender, S., Ed. *Distributed Systems*, second ed. Addison-Wesley, 1993.

[14] Object Management Group. *CORBAservices: Common Object Services Specification*, November 1997.

[15] Stardust Technologies, I. Introduction to ip multicast routing. Web Page, http://www.ipmulticast.com/community/whitepapers/introrouting.html. Accessed April, 1998.

[16] Startdust Forums. The IP Multicast Initiative. Web Page, http://www.ipmulticast.com. Accessed April, 1998.

[17] Stevens, W. R. *Unix Network Programming*, first ed. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1990.

[18] Stevens, W. R. *Advanced Programming in the UNIX Environment*, first ed. Addison-Wesley, 1992.

[19] Stroustrup, B. *The C++ Programming Language*, second ed. Addison-Wesley, 1991.

[20] Talpade, R., and Ammar, M. H. Single connection emulation (SCE): An architecture for providing a reliable multicast transport service. College of Computing, Georgia Institute of Technology, Atlanta, GA 30332.

[21] Tanenbaum, A. S. *Computer Networks*, third ed. Prentice Hall, Upper Saddle River, New Jersey 07458, 1996.

[22] Visigenic Software Inc. *Visibroker for C++ Programmer's Guide*, October 1996.

[23] Visigenic Software, Inc. *Visibroker for C++ Reference Guide*, October 1996.