UNIVERSITY OF NEWCASTLE UPON TYNE
DEPARTMENT OF COMPUTING SCIENCE

A MIDDLEWARE SERVICE FOR FAULT-TOLERANT GROUP COMMUNICATIONS

Ph.D. THESIS

BY

Graham Morgan

NEWCASTLE UPON TYNE

SEPTEMBER 1999

Abstract

Many distributed applications require multicast group communication services, enabling an entity to interact with a group of other entities. Providing the reliability and ordering guarantees required by group based applications is not a trivial task in distributed systems where computation and communication delays might not be known accurately. Furthermore, the approaches available to support these guarantees are diverse. The choice of approach may significantly effect the performance of an application and/or may not be suitable for some application types.

Nowadays, distributed applications are frequently built as a Middleware service. The Thesis develops techniques for providing group communication support in Middleware environments. A group communication service has been designed and implemented in such a way as not to hinder the interoperability/portability of applications built using it. The service provides a variety of functions that may be tailored to suit many different types of applications.

Group communication protocols are presented that ensure reliability and ordering guarantees. Furthermore, the reliability and ordering guarantees of such protocols may be tailored to suit a wide variety of applications. Mechanisms that provide a variety of approaches to inter-member and inter-group interactions that are suitable for satisfying the requirements of many different types of applications (e.g., fault-tolerant, collaborative) are also supported. The service can work over local and wide area networks (Internet).

Acknowledgements

I would like to express my sincere gratitude to several people who have contributed in various ways to the completion of the Thesis.

First and foremost, I thank my supervisor, Professor Santosh K. Shrivastava. His guidance and technical contributions throughout this work have been invaluable. The enthusiasm and encouragement Santosh consistently showed throughout my PhD studies have been essential for the completion of the Thesis.

Many thanks to Hewlett Packard laboratories in Bristol (United Kingdom) and the Engineering and Physical Science Research Council (United Kingdom) for providing financial support for the research presented here. Special thanks to Mr. Roger Flemming for his contributions and advice during my stays at the Hewlett Packard research laboratory in Bristol.

I am indebted to Dr. Paul Ezhichelvan for the countless discussions I have had with him relating to work presented by the Thesis. I declare that the research presented by the Thesis relating to group membership and total ordering is partly the work of Dr. P. Ezhichelvan and Dr. R. J. de A. Macedo. Their earlier research efforts made the Thesis possible.

I would like to thank colleagues who have helped in various ways during my Ph.D studies, in particular Dr. M. Little, Dr. S. Wheater, Dr. F. Ranno, Dr. C. Angus, Dr. J. Steggles, Dr. D. Nelson, Mr. R. Smith, Mr. B. Arief, Mr. I. Welch, Mr. A. McGough, Mr. M. Beet, Mr. A. Garmew and Mr. L. Lloyd.

Finally, I would like to thank my wife, Tanya, and my family for their help and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Networked computer systems offer several advantages:

- *High availability of services* - It is possible to structure a network of computers in such a way that there exists no single point of failure. This enables a network of computers to endure a degree of component failure (computer and/or part of the network) and still carry out computational tasks (be it in a full or restricted manner). In a centralized system, failure of the computer will result in complete failure, with no further completion of computational tasks.
- *Collaborative tasks* - Organizations are by their nature distributed, with individuals working in different locations but nevertheless requiring the ability to exchange information. Geographically associating computers to users ensures that only computational tasks that require inter-computer communication may endure network delays, whereas other tasks may be carried out on a user's computer without the need to endure communication delay overheads. In a centralized system (consisting of a central computer and distributed "dumb" terminals) all computational tasks must endure network delay.

Applications that are implemented in a distributed manner over a computer network are commonly termed *distributed applications*. The development of distributed applications is made difficult if a variety of operating systems and communication protocols are involved. This is because computer networks tend to be heterogeneous in nature (many different types of platform may exist) with propriety service application programming interfaces (API) present on a per-platform basis.

Middleware systems present an application developer with services that ease the development of distributed applications. These services shield the application developer from platform specific type services; a programmer may develop applications wherever suitable Middleware services exist, irrelevant of underlying propriety services. To ensure different vendors provide Middleware services that interoperate, standards are required; services are defined via a standard interface language and communications between them is accomplished via a standard protocol.

Currently there is no support for a *group communications* service in Middleware standards. A group communication (commonly termed *multicast*) describes the sending of a message to multiple recipients simultaneously. A group communication service would be highly desirable: applications exist that have requirements more suitably satisfied by such a service than a one-to-one (*unicast*) type

communication service. Providing a group communication service that may satisfy a wide variety of application requirements within a Middleware environment is the subject of the Thesis.

1.1 Distributed Systems

A distributed system may be characterized by the nature of its interconnection network. [Bal90] uses the terms tightly coupled and loosely coupled to differentiate between types of distributed systems. These two types may be described as follows:

- *Tightly coupled* - Processes are physically close to one another and communication is fast and reliable. Examples of such systems are hypercubes [Ranka88] and Transputer networks [May84]. Communication times are measured in microseconds.
- *Loosely coupled* - Processors are physically dispersed and communications can be relatively slow and unreliable. An example of the communication and processor provision for these systems is a workstation local area network (LAN). Communication times may be measured in milliseconds. However, in a wide area network (WAN) this can rise to over ten/one hundred milliseconds and in long haul networks (e.g., Internet) this time could be more reasonably measured in seconds.

From the above descriptions, loosely coupled is the term that best describes the systems that are the concern of the Thesis.

1.2 A Group Communication Service

A group is defined as a collection of distributed entities (objects, processes) in which a member can communicate with other members by sending messages to the full membership of the group (*multicasting*). With a group is usually associated a name to which applications may refer, making transparent the location of the distributed entities forming the group. This section continues with descriptions of how entity failure, message ordering guarantees and overlapping groups make the provision of group communications a non-trivial exercise.

1.2.1 Failures and Group Membership

A desirable property is that a given multicast be *failure atomic*: if a process crashes while multicasting a message, either all or none of the functioning members deliver the message. It is worth noting at this point that group communication protocols make a distinction between the receiving of a message and the delivery of such a message to the application layer (see 2.5.2). An additional property of interest is guaranteeing *total order*: all the functioning members are delivered messages in identical order (each member delivers the same set of messages in the same order). As an example,

these properties are ideal for replicated data management for high availability: each member manages a copy of data, and given atomic delivery and total order, it can be ensured that copies of data do not diverge. However, as discussed below, achieving these properties in the presence of process and network failures is not simple.

Assume that group members could be geographically widely separated, say communicating over the Internet. Therefore the communication environment is assumed such that message transmission times cannot be accurately estimated, and the underlying network may well get partitioned, preventing functioning members from communicating with each other.

Suppose that a multicast is interrupted due to the crash of the member making the multicast; this can result in some members not receiving the message. Member crashes should ideally be handled by a fault tolerant protocol in the following manner: when a member does crash, all functioning members must promptly observe that crash event and agree on the order of that event relative to other events in the system. In the type of environment considered here this is impossible to achieve: when members are prone to failures, it is impossible to guarantee that all functioning members will reach agreement in finite time [Fischer85]. This impossibility stems from the inability of a process to distinguish slow members from crashed or disconnected ones. One way to circumvent this impossibility result is to permit processes to suspect process crashes (sometimes incorrectly) and to reach agreement only among those processes that do not suspect each other [Chandra91, Schiper93]. This leads to a membership service which ensures that the functioning members that do not suspect each other install an identical sequence of membership views, with each view installation being identically synchronized with respect to message delivery events.

1.2.2 Message Ordering

There are instances when multicast messages need to be delivered in such a way that some specific order is not violated. As mentioned in the last section (see 1.2.1), total ordering together with failure atomicity are ideally suited to the management of replicated data. *Causal* ordering [Lamport78] is another type of message ordering that may be required within a group. Causal ordering guarantees that messages are delivered with respect to any causal relationship that may exist between messages; If message m could have caused message m' then every member of a group should delivery m before m'.

As an example of the need for causal ordering consider the following. Three users (*U1*, *U2*, and *U3*) participate in a text based conference via a distributed conferencing application. Messages sent by a user are multicast to the full membership of the group. *U1* multicasts a message *m1*. *U2* receives *m1* and multicasts *m2*. *U3* then receives *m2* followed by *m1*. *U3* may not understand the conversation occurring between *U1* and *U2* as it appears (to *U3*) that *U2* sent *m2* before *U1* sent *m1*. To ensure discussions are understandable by all users messages should be causally ordered.

There are two distinct types of protocol for achieving total ordering:

- *Asymmetric* - A single member of the group is responsible for ordering.
- *Symmetric* - All members of the group share the responsibility for ordering.

In an asymmetric scenario the member responsible for ordering is commonly termed the *sequencer*. Each member of a group may unicast the message they wished distributed throughout the membership of the group to the sequencer. The sequencer is responsible for multicasting such messages to all members of the group. In a symmetric scenario a member simply multicasts their messages to the whole membership of the group.

The choice of asymmetric or symmetric ordering techniques can affect the performance of a distributed application that makes use of group communications; symmetric ordering favors Groupware applications whereas asymmetric ordering favors highly available applications [Morgan99] (for descriptions of these different types of applications see 1.3).

1.2.3 Overlapping Groups

In some applications, entities are required to simultaneously participate in multiple groups [Birman91c, Garcia-Molina91]. For example, a computer based conferencing application may allow users to participate in a number of conferences simultaneously. Guaranteeing message ordering across groups is not trivial as the following example shows:

An entity *X* that is a member of multiple groups generates a message *m2* in a group *B* as a consequence of a message *m1* delivered to *X* in group *A*. Other entities that also participate simultaneously in groups *A* and *B* would then be required to deliver *m1* before *m2*, even though they originated in different groups. There must be a way of informing members of both *A* and *B* of the causal relationship between *m1* and *m2*.

1.2.4 Clients and Server Groups

The provision of a service may be accomplished via a group of objects/entities. Such a group is termed a server group. Clients may gain service from a server group (issue requests and receive replies). A server group may provide a service to clients via an open group or closed group approach:

1. *Closed group* - A client is considered a member of the server group and multicasts requests to each member of the server group. When message latency is high between a client and a server group (e.g., geographically separated by large distances) client requests will take far longer to service than if the server group was a singleton. Furthermore, transport level multicast may not be available due to the heterogeneous nature of the networks that connect a client to a server group

(this is the case when communications between a client and a server group is enabled via the Internet). As a member of the server group, a client may be required to participate in group communication protocols as a member of a server group (e.g., group management, message ordering), possibly requiring further multicasting on behalf of the client and possibly the delaying of messages until ordering guarantees are satisfied. For this reason, closed groups are more appropriate when clients and a server group exist on the same LAN or neighboring LANs.

2. *Open group* - A client is not considered a member of the server group and issues requests to just a single member of the server group. Unlike the closed group, clients do not participate in group communication protocols as a member of the server group. This makes the open group approach more suitable than the closed group approach for use in wide area networks (WANs), such as the Internet, when message latency between a client and a server group may be high.

1.2.5 Summary of Group Communications

To summarize, the aim of a group communication service is to aid communications between entities that collaborate to perform a task. To achieve this, the following mechanisms are required:

- *Multicast* - a single entity to send a message to multiple entities simultaneously.
- *Ordering* - a protocol to guarantee total message ordering within a group. Other ordering guarantees may be required (see 2.5.2) but total ordering is a necessity to enable members of a group to attain mutual agreement of group membership.
- *Group membership* - Enable members of a group to agree on group membership.
- *Ordering protocol* - message ordering may be achieved via asymmetric or symmetric ordering protocols.
- *Closed and open groups* - a client may interact with a server group via a closed or open group approach.

1.3 Use of Group Communications

Applications that require fault-tolerant services (*highly available* applications) and collaborative services (*groupware* applications) are the prominent application types driving the development of group communication services. This is because each of these applications are commonly structured as co-operating processes/objects over a computer network. The following sections describe the relevance a group communication service has for these two types of applications.

1.3.1 Collaborative Services

Groupware applications are primarily concerned with the sharing and presentation of information for groups of users. This information may be presented in a visual way, textural way, via audio or a combination of these. For example, users participating in a teleconference expect to see images of other users and hear what they are saying. Furthermore, such applications may present users with a mechanism for distributing textual messages throughout the group. The benefits of a group communication service to a groupware application are:

- *Multicast* - Information (visual, audio, textual) needs to be sent from a single user to multiple users simultaneously.
- *Message ordering* - The understanding of information by users should not be inhibited by the order with which such information is presented to users.
- *Group membership* - Users may maintain 'up to date' views of the participating user list.
- *Ordering protocol* - Experiments suggest that symmetric style protocols tend to perform better than asymmetric protocols in collaborative services [Morgan99]. Therefore, symmetric style protocols should be available.
- *Closed and open groups* - The closed group approach ensures that users are aware of all other users in a group. An open group approach may hinder this awareness as a client only interacts directly with a single group member.

1.3.2 Highly Available Services

A common method used to increase the availability of a service is to replicate the service over nodes in a network. A service that is replicated is commonly termed a *replica group*. The aim of a replica group is to allow the failure of nodes, parts of the network, or a number of objects that provide the service to be tolerated before the service becomes unavailable. When there is more than one replica in a group there is a possibility that different clients can make use of different replicas simultaneously, perhaps attempting to modify their states in a conflicting manner. If this occurs then different replicas may have different states, resulting in a number of replicas possibly providing contradicting information to a client request. A replica consistency protocol is necessary to ensure than concurrent invocations on different replicas leave all replicas in a mutually consistent state. There are two categories of replica consistency protocols:

- *Active* - Operations are invoked on all replicas of a given group. All replicas may return replies to the client.

- *Passive* - Operations are invoked on only one member of the replica group (the primary). This replica returns a reply to the client. To ensure that the states of the other (passive) replicas are updated, the primary checkpoints its state to the other members of the group. Failure of the primary requires the passive members to participate in an election process that results in one of the passive members assuming the role of the primary.

The benefits of a group communication service to an application that makes use of replicas in an attempt to increase the availability of a service are:

- *Multicast* - Enable client invocations to be sent to all replicas (active). Allow a primary to send values relating to its state to passive replicas (passive).
- *Message ordering* - Client requests are delivered in a total order to prevent state divergence (active). Aid the provision of failure atomicity (total ordering) when enacting a protocol to elect a new primary.
- *Group membership* - Realise the failure of a replica (active and passive). In the case of primary replication, if a primary fails then the protocol used to elect a new primary may be initiated by any of the passive replicas.
- *Ordering Protocol* - Experiments suggest that asymmetric protocols tend to perform better than symmetric protocols when used to support a replication protocol [Morgan99]. Therefore asymmetric protocols should be available.
- *Closed and open groups* - Open groups enable a client to perceive a replica group as a singleton (the replication becomes transparent to the client). This is not the case with a closed group, as a client is aware of all replicas.

1.4 Middleware

A remote procedure call (RPC) [Birrell84] is commonly used primitive for communication between entities of a distributed application. In essence, a client may access the services of a remote server in a similar way as a client may access the services of a function within its own addressable space, via a procedure call. However, such a call actually results in inter-process communication (possibly over a network). The mechanisms that enable such communications are hidden from the application developer.

Common Middleware services have extended this notion of inter-process communication to that of inter-object communications, allowing programmers to apply object-oriented programming techniques to their distributed applications; objects are regarded as the unit of distribution. Examples

of such Middleware are Sun Microsystems' Java-Remote Method Invocation (Java-RMI), Microsoft's Distributed Component Object Model (DCOM) and the Object Management Group's Common Object Request Broker Architecture (CORBA).

1.5 Contributions of the Thesis

This Thesis addresses the problems related to the provision of group communications for use by middleware services. In particular, we attempt to identify and support group communication mechanisms that may benefit both groupware and fault-tolerant applications.

The NewTOP protocol suite [Ezhilchelvan95] is described. These protocols provide a number of ordering protocols (arbitrary, causal and total) across overlapping groups. Furthermore, protocols for the management of group membership are provided allowing multicasts to be failure atomic, ensuring members maintain a mutually consistent view of group membership. The protocols provided by NewTOP are not restricted to any type of computer network allowing NewTOP to operate on networks as diverse as LANs and the Internet. Symmetric and asymmetric ordering protocols may be used on a per group basis. Multi-group members may use symmetric ordering in one group while using asymmetric ordering in another.

Protocols are described that are suitable for enabling clients to access a service provided by a group (*server group*) in an open or closed manner. Clients gain service by invoking operations on the server group. Invocations may be synchronous or asynchronous. These protocols are then extended to allow situations when a client issuing requests at a server group may be a group itself.

The mechanisms and protocols described in the Thesis are implemented as the NewTOP service. The NewTOP service is provided as a CORBA service. Implementation issues of the NewTOP service are described. Experiments involving the NewTOP service are described and the results of such experiments presented. The analysis of the results gained from these experiments aim to identify suitable protocol configurations for highly available and groupware applications.

The novelty/contribution of the Thesis may be summarized as follows:

- *Implementing a Middleware service that supports object groups (chapter 4)* - The CORBA environment has been extended to support group communications in a standard way. This ensures that applications built using the NewTOP service may benefit from the interoperability benefits associated with CORBA.
- *Allowing clients of an object group to use either an open or closed group approach (chapter 5)* - The ability of the NewTOP service to support overlapping groups and ensure total ordering of messages over such groups has been exploited to provide open and closed client/server group invocation protocols.

- *Analyzing the performance of the object group service that is described in chapter 4 (chapter 6)* - These experiments were devised to determine the performance of protocol configurations that may be appropriate in the support of highly available and groupware applications over LAN and WAN environments.

1.6 Thesis Outline

The Thesis is structured as 7 chapters. Background material is presented in chapter 2. Chapter 3 describes the group communication protocols of the NewTOP protocol suite. Chapter 4 describes the implementation of these protocols via the NewTOP service. Chapter 5 describes the invocation protocols that enable clients to issue requests for server groups. Chapter 6 presents results from a number of experiments carried out using the NewTOP service. Finally, chapter 7 gives the conclusions and possibilities for future work arising from the work presented in the Thesis.

Chapter 2

Background

In this chapter the relevant issues related to the provision of an object group communication service for use in Middleware environments are described. A description of the most popular of these Middleware technologies is presented. The mechanisms that may be required to provide a group communication service are described. Finally, recent research related to group communications support for Middleware is presented and the novelty of the work described in the Thesis is identified.

2.1 Middleware

Middleware systems present an application developer with services that ease the development of distributed applications. These services shield the application developer from platform specific type services. The term platform, as used here, indicates low level services and processing elements defined by processor architecture, an operating system's application programming interface (API) and communication primitives (such as sockets for inter-process communication). This section defines Middleware technologies via the properties they seek to exhibit and describes the mechanisms that are common in the enabling of such technologies.

2.1.1 Properties of Middleware

Middleware is commonly understood to mean the layer that sits between applications and operating systems, its function is to ease the development of distributed applications. Existing literature [Bernstein96] characterize Middleware by four properties:

1. *Platform independence* - Implementations should be available over a number of different operating systems and hardware configurations. The types of available platforms should not hinder an application developers use of a Middleware.
2. *Distribution* - Capable of providing support for services that are not restricted, geographically, to a single location.
3. *Provision of standard interfaces and protocols* - Irrelevant of implementation environment, protocols used by Middleware and interfaces to the Middleware remain the same. As networked systems tend to be heterogeneous in nature, propriety service APIs may be present on a per-platform basis. A developer that becomes proficient on one platform may not be able to easily

transfer his/her skills to another type of platform. To avoid this, a Middleware technology should seek to provide an API that does not deviate, irrespective of which platforms it resides.

4. *Generic* - Meets a wide variety of application requirements across many industries.

The *client/server* model is commonly used in Middleware in an attempt to satisfy property 2 and property 4. This is because clients and servers are suitable units for distribution (property 2) and have been used to build a wide variety of application types (property 4). In the client/server model a server satisfies the service requirements of one or more clients. A service is defined by an interface. This interface indicates to a client the manner of interaction a client must assume to gain service from a server. Interaction is realized by a client issuing requests via a suitable message passing protocol and the server replying to client request via the same protocol. Achieving property 1 and property 2 has proven more difficult. This is due to the variety of low level services and the large number of different organizations involved in the development of such services. Organizations have to cooperate to formulate and agree on industrial standards and adhere to them to ensure that programming APIs and protocols remain consistent over various platforms, irrelevant of the vendor supporting the Middleware implementation. This section continues with a description of a method commonly used to implement the client/server model in Middleware.

2.1.2 The Proxy/Stub Method

The proxy/stub method has been used for more than fifteen years to enable distributed application development. Most notably, this method is used to enable a remote procedure call (RPC) [Birrell84]. In essence, a proxy resides in the same address space as a client and presents the client with an interface to a service. This interface is presented in a manner that would suggest to the client that this service is no different than any other service located within its own address space. However, requests directed at the proxy interface are then forwarded, by the proxy, across process boundaries, and more usually a network, to the actual service implementation. At the service side a stub, located in the same address space as the server, is responsible for receiving these requests and then forwarding them to the service implementation, receiving any replies, and then returning these replies to the proxy. Replies received by the proxy are then returned to the client.

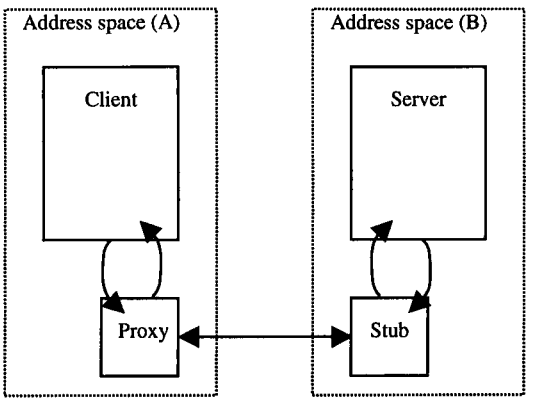

Figure 2.1 - Client server implementation using stubs and proxies.

Simply stated, mechanisms involved in the implementation of the proxy/stub method hide from the application developer many of the technical details involved in passing information across process boundaries and networks. For example, ensuring data types are structured in a suitable manner for passage over a network (marshaling) and managing low level transport protocols.

2.1.3 Distributed Objects

There are three dominant types of Middleware that provide an object-oriented approach to distributed application development:

- Sun Microsystems' Java with Remote Method Invocation (Java-RMI) [Sun97]
- Microsoft's Distributed Component Object Model (DCOM) [Brown96]
- The Object Management Group's Common Object Request Broker Architecture (CORBA) [OMG95a]

To ease the production of proxies and stubs for use within distributed applications, Java RMI, DCOM and CORBA provide:

- A language for defining an interface of a service.
- Some mechanism for automating stub/proxy generation from an interface definition.
- A method, appropriate to the target languages of the client and server, for integrating proxies and stubs into client and server code.

More detailed descriptions of Java-RMI, DCOM and CORBA are presented in the following section.

2.2 Object-Oriented Middleware Technologies

The purpose of this section is to describe Java-RMI, DCOM and CORBA. Due to the substantial subject areas each of these may cover, a simplified view of the processes required to produce proxy/stub code and the enabling of inter-object communication via this code is described. At the end of this section these technologies are compared to the original four properties that aid in defining Middleware (see 2.1.1).

2.2.1 Java-RMI

The Java programming language enables application developers to write object-oriented programs that may be executed on a variety of platforms without alteration. This is achieved via an environment that provides a consistent API within which a Java program may execute. This environment is termed the *Java virtual machine* (JVM).

Once written and compiled, Java code will work wherever a JVM exists. However, it was not until 1997 that support was added to the Java language that enabled objects in different address spaces to communicate using the proxy/stub mechanism. This support took the form of the Java Remote Method Invocation (Java-RMI).

The use of Java-RMI is specified within the Java language definition and a simplified view of its use is thus:

1. *Specify a service using a Java interface* - An interface in Java is simply a construct that defines methods, like a class, but does not provide any implementations for these methods.
2. *Implement interface* - It is necessary to implement a service interface with a class.
3. *Create server program* - A program that acts as the "server" is required to create an instance of the service defined by an interface and makes the interface available for remote clients.
4. *Create proxy and stub code* - The Java development environment supplies a tool, *rmic*, that creates the proxy and stub codes automatically from the original interface definition of the service.
5. *Publicize server to clients* - There is a service, the registry service (supplied with the Java development environment), that is required to be run to enable clients to find servers and retrieve the appropriate stub code to use a remote service. Servers register all the information required by clients to enable clients to contact them via the registry service.
6. *Enable client server interaction* - A client retrieves a reference (in reality the proxy code) from a server via the registry service.

Fig. 2.2 shows how an interface may be used to define a simple service that takes two numbers and returns their sum. The interface is easily understood and trivial to develop. By inheriting from the interface "java.rmi.Remote" the appropriate functionality is included to enable networked communications (which are hidden from the application developer). A further addition needed for a remote service is that all methods must be declared to throw a "java.rmi.RemoteException". This is because many various exceptions may be thrown due to communication failures, such as timeouts and crashed servers, which do not occur in single address space application development.

```
Public interface AddUp extends java.rmi.Remote {
        int addItems(int x, int y) throws java.rmi.RemoteException;
}
```

Figure 2.2 - A Java interface suitable for specifying a remote service.

Java-RMI is designed to work when clients and servers are implemented in Java. There is no support for clients and servers if they are implemented in other programming languages.

2.2.2 DCOM

Microsoft's DCOM, previously known as Network OLE (Object Linking and Embedding), is an extension of the COM (Component Object Model) designed to network applications. The underlying objective of COM is to permit the independent development of software components that can intercommunicate, regardless of language or function. The unit of distribution in the DCOM environment is commonly termed a component.

A component exports one or more interfaces that defines its functionality. Interfaces may be constructed in an object-oriented fashion, allowing application developers to make use of polymorphism and inheritance. A component consists of an array of function pointers, each pointer indicates the physical address of a method supported by the interface. By placing components into a dynamic link library (DLL) applications may link to (bring into their own address space) components at run time that are implemented in arbitrary languages. This increases code reuse and allows components to be distributed amongst applications. Examples of popular components are buttons that may be found on toolbars that form part of a user interface. The functionality and appearance associated with a button may be implemented in C++, BASIC or Java. Once the implementation details of the button have been encapsulated by a component the button may be used by programmers in many different types of applications written in any language (e.g., Visual Basic, Visual C++).

Extending COM to DCOM required the introduction of an interface definition language (IDL) that aided the production of proxies and stubs for use by clients and servers respectively. The IDL used by DCOM is based on the IDL standard specified by the Open Software Foundation (OSF)

for use with its Distributed Computing Environment (DCE) [Rosenberry92]. A simplified view of producing a service to be used remotely by clients in the DCOM environment is thus:

1. *Build the component that implements the service and register it with the windows environment.*
2. *Specify the service using IDL* - Include in this definition the identity of the component that implements the service.
3. *Pass the IDL that specifies the service through the Microsoft IDL (MIDL) compiler* - A number of different files (in the C language) are produced that provide the proxy/stub code.
4. *Compile and link the C code produced by the MIDL compiler* - The aim is to produce a proxy/stub DLL.
5. *Register the proxy/stub DLL with the windows environment* - Microsoft supply tools to enable this.
6. *Indicate to the windows environment where the remote service for a proxy/stub DLL resides* - Use the DCOM configuration tool to indicate to the windows environment on which machine in a network the service (together with an associated proxy/stub DLL) resides.
7. *Enable client/server interaction* - Clients may use the proxy DLL in the same way as any other DLL. However, care must be taken to ensure that the proxy DLL is on the same machine as the clients. The operating system is responsible for incorporating DLLs into application code once the DLL has been registered.

Producing components is not a trivial matter. Component and IDL interface design do not present as easily understandable object-oriented code as in C++ or Java. This is because DCOM is tightly coupled to the Microsoft operating system and therefore relies heavily on mechanisms that may not appear intuitive to a programmer that is not well versed in the Microsoft environment. To exemplify this an interface that provides the same functionality as the Java-RMI "AddUp" example is shown in fig. 2.3.

```
[
        object,
        uuid(32bb8324-b41b-11cf-a6bb-0080c7b2d682),
        helpstring("AddUp Interface"),
        pointer_default(unique)
]

interface AddUp : IUnknown
{
        HRESULT addItems([in] int x, [in] int y, [out] ans);
};
```

Figure 2.3 - A DCOM IDL interface suitable for specifying a remote service.

The interface in fig. 2.3 requires information additional to that described in the Java example (fig. 2.2), such as specifying the component identifier for this interface and what type of pointer manipulation is required for marshaling purposes (unique in this case - can be NULL, change in the function but cannot be made into aliases within the function). In the function definition parameters have to be specified as *in* or *out*. This aids the marshaling process when determining suitable memory allocation for message structures. For example, if all parameters are classed as *in* then there is no need to allocate message space or even marshal them for the reply message. The answer is returned as an out parameter and not as the value of the function. This is because the return value of a function contains information regarding the success of the remote call. If there are any exceptions they are returned via this value (HRESULT).

Programming tools (Microsoft's Visual C++ 6 and Visual Java 6) have automated the production of components and their distribution with easy to use mechanisms. A developer simply creates a class in the normal way associated with the programming language being used, and then indicates to the programming tool that this class is to be turned into a component. IDL generation, MIDL parsing and component registration within the windows environment are all taken care of by the programming tool.

2.2.3 CORBA

The Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) is a widely accepted standard for Middleware. Over 700 companies endorse the standard with implementations of the standard existing on most operation systems.

Objects may interact irrespective of the languages used for their implementation; service providers may be implemented in C++ whereas clients of such a service may be implemented in COBOL. This interoperability is achieved by ensuring all service providers specify the services they provide via a standard language (IDL). Unlike DCOM, the IDL language used by CORBA resembles a more traditional object-oriented language (C++). CORBA and DCOM approach the issue of separating implementation from interface in the same manner; via an IDL.

To produce the appropriate proxy/stub code required to enable clients and servers to interact, an IDL interface is passed through a parser (supplied by a vendor). The language of the code produced depends on the parser. Most parsers accommodate C++ or Java. There is support for other languages, such as COBOL and C. The proxy/stub code produced implements a layer of abstraction known as the Object Request Broker (ORB) within the CORBA environment (see 2.3.1). To ensure cross compatibility over different platforms between IDL and target languages the CORBA standard specifies mappings from IDL data types to data types found in various languages. A simplified view of the production of a (possibly) remote service is thus:

1. *Specify a service using CORBA IDL*
2. *Create Proxy and stub code* - Pass the interface through a parser supplied by a vendor.
3. *Implement the service* - The object that implements the service is written in the same language as the proxy/stub code. Most parsers present developers with "ready to use" skeleton code suitable for implementing the service.
4. *Write a server program to support the service* - The server program creates an instance of the object that implements the service and activates the required mechanisms within the CORBA environment to ensure communications between the service and clients may occur.
5. *Publicize server to clients* - A mechanism is required to enable clients to retrieve a reference to the service. This may be done via the *Naming Service* (a service where clients can request services by a well known name and retrieve appropriate service references). Alternatively, service references may be cast into the form of a string and passed to a client by other methods (email, command line parameters, etc.).
6. *Create client* - A client is created with the appropriate proxy code included in the client source code.
7. *Enable client/server interaction* - Clients retrieve the object reference of a desired service (either via the naming service or by other means). Once a reference is retrieved communications between client and server may commence.

```
interface AddUp
{
        exception addFailure {string reason;};

        short addItems(in short x, in short y) raises addFailure;
};
```

Figure 2.4 - A CORBA IDL interface suitable for specifying a remote service.

Fig. 2.4 shows the AddUp service specified by a CORBA IDL interface. Notice, as with DCOM, parameters must be defined as *in* or *out* to aid parameter marshaling. However, unlike DCOM, it is possible for functions to have value. Exceptions are treated in a similar way as in Java RMI. Developers may define their own exceptions and then apply them to functions via the "raises" mechanism. Any call to a remote object, irrespective of the existence of user defined exceptions, may raise an exception and be handled by the client that enacted the function call.

2.2.4 Evaluation of Surveyed Middleware

Java-RMI, DCOM and CORBA will now be evaluated with respect to the four desirable properties of Middleware (see 2.1.1).

- **Platform independence**
  - *Java-RMI* - The presence of the JVM on most operating systems ensures that platform dependency is all but removed from issues relating to Java application development. However, it is assumed that Java is the only language that a developer may wish to use. Although dependency is removed from low-level platform services (operating system, network architecture, etc.), application dependency is now placed on the JVM.
  - *DCOM* - Support for DCOM, and therefore COM, development is well supported on Microsoft operating systems. Microsoft operating systems (such as Windows95, Windows98, and WindowsNT) specify their APIs with the aid of COM ensuring that application developers familiar with Microsoft programming environments understand COM and therefore produce their own COM objects. As Microsoft operating systems are more wide spread than any other, there exists a wealth of applications for them. This has resulted in the existence of many COM objects ready made for use by application developers (such as interface components). Unfortunately, due to the tight coupling of DCOM and COM with the Microsoft environment, presenting DCOM and COM on other platforms has proved difficult and for this reason is not widespread on non-Microsoft environments.
  - *CORBA* - CORBA is the industry standard for specifying Middleware services. This has allowed many vendors to implement services that are capable of working over a large number of platforms. Application developers may develop CORBA services in a number of programming languages on many different operating systems using tools supplied by many different vendors with the knowledge that such applications may interoperate.
- **Distribution** -
  - As already mentioned, Java-RMI, DCOM and CORBA incorporate distribution into their architectures via the proxy/stub method.
- **Provision of standard interfaces and protocols** -
  - Java-RMI - As Java is the only language used the interface to Java-RMI remains consistent. The protocol used to enable Java-RMI is a propriety protocol built on top of TCP/IP. This is adequate to enable JVM to JVM communication but does not allow communications with non-JVM environments. To overcome this, the latest release of the JVM (1.2) allows a developer to specify the use of a different protocol to enable communications with CORBA objects.
  - DCOM - The IDL standard supplied by the OSF for defining services coupled with the language independent way COM objects may be specified ensures that all services may be defined in a consistent manner in the Microsoft environment. The protocol used for communications is a RPC protocol, originally specified by the OSF for use within DCE. As

the protocol and the IDL standards are taken from existing technologies (DCE) there is scope for other vendors to produce tools that enable the communication of DCOM objects with objects on non-Microsoft platforms that are also enabled via DCE interfaces and protocols. However, services based on DCE were never widely implemented and so other cross platform solutions are provided for use with DCOM. Most notably CORBA/DCOM bridges are available which enable DCOM objects and CORBA objects to communicate.

- CORBA - CORBA is a standard. This ensures that the IDL and the protocol that enables inter-object communication, commonly termed the Internet Inter-ORB Protocol (IIOP), remain consistent, irrelevant of implementation. However, due to the large number of vendors that are involved, additional propriety protocols and interface extensions have appeared [Iona95] [OOC98]. Services that rely on such extensions may be unavailable to clients that have been developed on ORBs that do not have access to such extensions.

- **Generic -**
  - Java-RMI, DCOM and CORBA all enable communication via an RPC, with all details of transport protocol services hidden. This is adequate for the majority of applications, however, there exists applications that may be more suited to other types of transport services not provided by existing Middleware. For example, teleconferencing requires a video image to be sent to a large number of receiving clients on a regular basis. There are transport level protocols [Savets96] that are more suited to managing the transfer of large quantities of data between objects. This has been recognized by the OMG and there is scope in the next release of CORBA to allow transport protocols (other than IIOP) to be used by an application developer.

Java-RMI, DCOM and CORBA are very popular in the area of distributed application development. The WWW has ensured the success of Java, Microsoft has ensured the success of DCOM and the acceptance of CORBA by all major computing companies has ensured the success of CORBA. Deciding which Middleware, from these three, to choose when developing a distributed application is difficult, as many requirements (such as those relating to an application and that of the organization developing the application) have to be considered. For the purposes of the research presented in the Thesis I have chosen CORBA. The main reason for this choice is the acceptance CORBA has as an industrial standard and the availability of CORBA over many different platforms. The following section describes the CORBA standard in more detail.

2.3 OMG & CORBA

In the OMG's book "Object Management Architecture Guide" [Soley95], the OMG identified it's approach to distributed application development:

"To adopt interface and protocol specifications that define an object management architecture supporting interoperable applications based on distributed interoperable objects. The specifications are to be based on existing technology that can be demonstrated to satisfy OMG's Technical Objectives"

The OMG developed a conceptual model, known as the core object model, and a reference architecture, termed the Object Management Architecture (OMA). The OMA consists of four components: Object Request Broker (ORB), Object Services (OS), Common Facilities (CF), and Application Objects (AO). CF relate to object services that aim to satisfy quite specific application requirements (e.g., e-commerce, database management systems) and AO relate directly to applications. As the Thesis is concerned with the development of services that aid the development of a wide variety of application types, the ORB and OS components are of concern and will be examined in more detail in following sections.

2.3.1 Overview of The ORB

The core of the OMA is the ORB. The ORB is a communication bus for objects. Fig. 2.5 shows how the ORB and object services are commonly displayed.

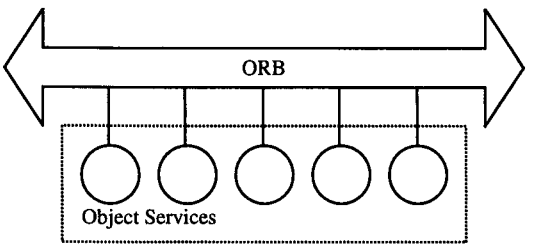

Figure 2.5 - ORB and object services.

The ORB architecture specifies an IDL for defining objects and a protocol, Internet Inter-ORB Protocol (IIOP), for enabling inter-object communications. IIOP is a protocol that specifies how detailed information representing a CORBA request is laid out on a network transport service. IIOP ensures multi-vendor interoperability between ORB implementations. Any functional enhancements to the ORB are achieved via object services. This ensures that applications will work on any ORB, irrelevant of the vendor supplying the ORB.

The IDL is simply a declarative language that supports no scope for programming implementation details. This is left to a programming language of the developer's choice. IDL is network neutral and operating system neutral, thus preventing developers from introducing platform dependent mechanisms into a service's IDL definition.

An Interoperable Object reference (IOR) is used to uniquely identify objects in CORBA. An IOR is a sequence of object-specific protocol profiles, plus a type identifier. The IOR is not intended to be visible to application programmers. Programmers are presented with a suitable structure available in the programming language of their choice to represent an IOR. For example, in the case of C++ this appears as a pointer to an object. As service interfaces defined in IDL may inherit other interfaces (also defined in IDL), mechanisms are provided that are capable of casting pointers to CORBA objects between class definitions in a similar manner as pointers to non-CORBA objects may be cast between class definitions.

The IDL allows an object reference to be passed as a parameter in a function call. This is the mechanism that enables the distribution of object references between objects. In addition to this mechanism, a developer may derive a string representation of an IOR and derive an IOR from a string representation. This is useful for passing object references by other methods (e.g., email).

2.3.2 CORBA Interface Architecture

The CORBA specification defines a number of interfaces to allow clients and servers to participate in inter-object communication. Following are descriptions of these interfaces:

- IDL Proxy - The IDL Proxy (sometimes termed IDL stub) presents an interface derived from an IDL definition of a service and is linked into the client program. IDL interfaces are commonly termed static as they define how a service may be used at compile time and may not change after compile time.
- IDL Stub - The IDL Stub (sometimes termed IDL skeleton) is simply the server side counterpart of the IDL proxy.
- Dynamic Interfaces - Statically including proxy/stub code derived from an IDL into client and server programs to enable inter-object communication satisfies the communication requirements for many applications. However, there are instances when this is not adequate. The static mechanism assumes clients are aware of servers and that servers are aware of the way they must satisfy client requests at compile time. This may restrict the way an application may evolve; allowing existing clients to use new services which are introduced during the lifetime of an application becomes difficult. To overcome this problem dynamic interfaces are supported by the CORBA standard.

- *Dynamic Invocation Interface* (DII) - Enables the specifying and building of requests at run time, rather than calling linked-in proxy code. Operations supported by the DII include: create_request, invoke, send, get_response. Invocations made by the static and dynamic methods are indistinguishable by the server object.
- *Dynamic Skeleton Interface* (DSI) - The server side analogue to the client side DII. The DSI inspects the parameters of an incoming request to determine a target object and method. This interface allows a service to assume the role of another service.

- ORB Interface - Enables direct access of the ORB by clients and servers. Only a few operations are supported, such as those for deriving IOR from strings and visa-versa.
- Basic Object Adapter Interface - The server program that supports the objects that implement services (defined by IDLs) is aided by the Basic Object Adapter (BOA). The server program registers objects ready for use with the BOA. Once this has occurred the BOA manages requests on behalf of the server's objects. Due to the fact that interaction occurs directly between an application and an ORB (and the possibility of an application to be programmed in any one of many languages), defining this interaction was left to vendors. This has resulted in the presentation of these mechanisms in a number of different ways, making it difficult to port code from one vendor's ORB to another. To overcome this, the next release of the CORBA specification identifies a Portable Object Adapter (POA) that seeks to standardize direct application to ORB communications.

The diagram in fig. 2.6 indicates how the interfaces of an ORB are integrated and which interfaces interact with clients and which interact with servers.

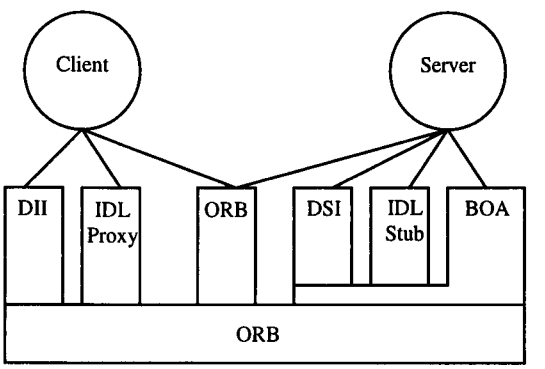

Figure 2.6 - The structure of CORBA interfaces.

2.3.3 Interface Repository

The interface repository [OMG95b] allows a service to register its interface. This enables clients to search for services at run-time. By using the interface repository, a client is able to locate an object that may have been unknown at compile time, enquire about its interface, and then issue requests via the DII to the newly created service. Furthermore, an existing service may identify an interface in the interface repository and assume the role of this service with the aid of the DSI.

2.3.4 Object Services

Following are brief descriptions of object services which are in common use:

- *Naming Service* - Supports name-to-object association. A hierarchical naming structure (similar to that used for UNIX file systems) has been adopted. This allows clients to retrieve the IOR of an object using a reasonable name. For example, an object that governs the bank account of a company may be registered as "/financial/bank/account/MicroSmart". The mechanism that enables a client to gain the IOR of the naming service is left to the ORB vendor.
- *Event Service* - De-couples the communication between objects. Objects may assume the roles of supplier or consumer. The service defines two approaches for initiating event communication: the *push model* and the *pull model.* A supplier uses the push model to transfer event data to consumers. Consumers use the pull model to request event data from a supplier. The event channel is simply an intervening object that allows multiple suppliers to communicate with multiple consumers simultaneously in an asynchronous manner.
- *Lifecycle service* - Represents a framework for creating, deleting, copying and moving objects. The creation facility is most commonly available in CORBA applications. As there is no existence of a basic creation facility in CORBA IDL (akin to a constructor in C++) *factory objects* are used to create instances of particular types of objects. There is no standard interface for a factory object, an application developer has to develop their own in a manner they see as appropriate. Usually, for each type of object there is a factory object.
- *Persistence Service* - Provides common interfaces to the mechanisms used for retaining and managing the persistent state of objects in a data store in an independent manner.
- *Transaction service* - Ensures that a computation of one or more operations on one or more objects provides properties of atomicity, consistency, isolation and durability (ACID properties).

2.3.5 Summary

After ten years and three release versions the CORBA standard is approaching a state of maturity with which organizations are showing increased confidence. This is shown by the drive in industry to

migrate existing legacy systems to CORBA environments [Morin98]. The service approach ensures that the interoperability of applications may be maintained irrelevant of any new functionality that may enhance an ORBs service provision. However, CORBA, JAVA-RMI and DCOM standards only support point-to-point (one-to-one) communications. Applications wishing to make use one-to-many communications (object groups) are not yet accommodated. The OMG is starting to address this issue [OMG98a]. This chapter continues by identifying the types of applications that may benefit from a service that may enable an application developer to make use of object groups.

2.4 Application Requirements

This section identifies groupware applications and applications that are required to exhibit a high degree of availability as two benefactors of object groups. Firstly, a service that provides object groups via group communication mechanisms is defined via the properties such a service aims to satisfy. This is a brief description designed to aid in the understanding of application requirements. A more detailed description of the properties of a group communication service are presented later in this chapter (2.5). By referring to these properties, the necessity for object groups to enable groupware and highly available applications are described.

2.4.1 Properties of a Group Communication Service

The term "group communications" infers the collaboration of entities/objects to perform tasks via messages directed at multiple recipients (group of entities/objects), rather than at a singleton. This type of message passing, one-to-many, is commonly termed a *multicast.*

In addition to providing a multicast mechanism systems that depend on group communications may also require quite sophisticated protocols to manage message delivery. For example, messages to be delivered at each member of a group in the same order. Furthermore, groups may be dynamic; members may leave and join a group during the lifetime of a group. To enable a multicast to be directed at actual members of a group (not including departed members), a mechanism is required which ensures that all members of a group have a mutually consistent view of group membership. A mechanism of this type is usually called a *group membership service.*

To summarize; A service that provides developers with mechanisms that support the integration of group communications into a distributed system should consist of the following:

- A multicast mechanism.
- Protocols for managing message delivery, with certain ordering and reliability properties.
- A group membership service.

A service that provides these properties will be referred to by the Thesis as a group communication service. This chapter continues with descriptions of distributed applications that may benefit from the presence of a group communication service.

2.4.2 Collaborative Applications

There is a steady increase in the popularity of computer applications that enable groups of people to work productively together. These applications (commonly referred to as Groupware) are perceived as replacing more traditional methods of working, such as face-to-face meetings, paper mail and telephone conversations. Groups of individuals that benefit from the use of these applications tend to be geographically seperated, making regular physical meetings difficult. Popular applications that may be considered groupware are: conferencing applications, Internet Relay Chat (IRC), bulletin boards.

Groupware applications are primarily concerned with the sharing and presentation of information for groups of users. A simple text based conferencing system that has three participants can be used as an example to show the important role a group communication service may provide within a groupware application:

A conference consists of a number of users that share information by sending messages to each other. A user may join a conference and leave a conference at any time. When a user wishes to send a message to a conference a single copy of the message is created and a single send operation is enacted by the user. If a participant in the conference receives a message, then all participants in the conference should receive the same message. As a result of receiving a message *M1* from user *U1* another user (*U2*) may wish to respond to the content of *M1* with a message *M2* (*M2* is commonly known as a *follow-up* message). A user (*U3*) that receives *M2* may be unable to understand the content of *M2* if *U3* has not yet received *M1*. Furthermore, *U3* may respond to *M2* before receiving *M1*, making discussions within the conference difficult, misleading and ultimately unproductive. Therefore, all participating users should receive a follow-up message after they have received all the messages that the follow-up message may relate to. In our example, this results in everybody receiving *M1* before receiving *M2*. The type of relationship between messages that our example highlights is commonly termed a *dependent relationship* or a *causal relationship*; *M2* depends on *M1*, *M2* is caused by *M1*. When ordering of messages is accomplished with respect to causality messages are said to be causally ordered.

Our example identifies the three requirements that a group communication service aims to satisfy in the following manner:

- *Sending a single copy of a message to multiple recipients via a single send operation* - A multicast mechanism is required to enable a user to send a single message to multiple users simultaneously.
- *Ensuring messages are presented to end users in an order that does not inhibit the understanding of discussions within the conference* - Protocols that preserve the causal ordering of messages and ensure reliable (all members of a group receive message) message delivery are required.
- *Managing new users and departing users* - To enable users to maintain "up-to-date" views of the participating user list of a conference a group membership service is required.

2.4.3 Highly Available Services

A common method used to increase the availability of a service is to replicate the service over nodes in a network. A service that is replicated is commonly termed a *replica group*. The aim of a replica group is to allow the failure of a number nodes, parts of the network, or a number of objects that provide the service to be tolerated before the service becomes unavailable. There are two main techniques available for providing service replication:

- **Active Replication**

  In active replication client requests are directed at each replica. Each replica then attempts to process the request and may reply to client requests. The active replication of an object requires two conditions to be met:

  1. *Agreement* - All the non-faulty replicas of an object receive identical input messages.
  2. *Order* - All the non-faulty replicas process messages in an identical order.

  Therefore, if all the non-faulty replicas have identical initial states then identical output messages in an identical order will be produced by them (assuming that an action performed by an object on a selected message is deterministic). To ensure the message ordering requirement is satisfied suitable protocols must be available that can provide guaranteed identical ordered message delivery within the replica group.

  When member failures occur clients of an active replica group may not suffer from gaps in service. The only time an actively replicated group cannot service requests is when all replicas have failed, or the replica group is unreachable by a client due to network failures.

  Detecting member failures is required to satisfy the agreement condition. Inconsistent views of group membership may lead to the failure of client requests reaching non-faulty replicas and/or the inclusion by some members of faulty members in their group views.

- **Passive Replication**

  Passive replication requires only one member of the replica group, the *primary* (sometimes referred to as the *coordinator*), to receive, process, and reply to client requests. To ensure that members of the replica group stay mutually consistent the primary must send a checkpoint of its state to the passive group members, usually when the state of the primary has changed.

  In the event of the primary failing the remaining members use a protocol to elect a new primary, which then takes over the duties of the failed primary.

  As opposed to active replication, it is not necessary for computations performed by the replicated objects to be deterministic: state is imposed upon the passive members of the group by the primary guaranteeing that all members of the group will remain mutually consistent.

An example of a bank account service made highly available via active replication may be used to demonstrate the benefit a group communications service may bring to a fault tolerant applications:

Copies of a bank account *B* reside at three of a bank's branches. This degree of replication allows routine audit checks of an account to be carried out at a branch (making the inspected account unavailable for a short time) while still allowing access to the other two copies of the account. The account is accessible via an Automatic Teller Machine (ATM). Each branch has an ATM (*A1*, *A2* and *A3*). Each copy of the bank account should present the same balance whenever queried via an ATM. When a transaction is requested at an ATM, information relating to the transaction request are formulated into a single message and sent to all copies of the bank account. Each account then acts on the request and replies with a suitable answer. The requesting ATM takes the first answer only and discards the rest. Whenever an account balance falls below zero, bank charges are incurred.

Let us concentrate on the functioning of a single account held jointly by a husband and wife. The husband deposits \$50 (generating a message *M1*) at *A1* and then withdraws \$20 (generating a message *M2*), again via *A1*. We may identify *M1* and *M2* as being causally related and expect *M1* to be received by all copies of the account before *M2* is received. If this is not so, then some accounts may actually become overdrawn (there will be a time when some accounts would show a balance of -\$20) causing bank charges to be incurred.

We now extend our example and assume the wife is requesting a withdrawal of \$20 (*M3*) at *A2* at the same time the husband is at *A1*. As there is no causal relationship between *M3* and the other two messages (*M1* and *M2*) *M3* may be received at any time by the replica accounts. If *M3* arrives before *M1* then bank charges will be incurred, if *M3* arrives after *M1* bank charges will not be incurred. Therefore, we have a scenario, where some accounts may incur charges while others do not. To overcome this inconsistency, we must ensure that messages arrive at the same order at each account. This type of ordering is commonly termed *total ordering* (identical ordering while preserving causality).

In the same manner as our previous groupware example identified the three requirements that a group communication service aims to satisfy, so the observation is repeated for the highly available account example (for the sake of completeness passive replication is also mentioned):

- *An ATM is required to send a single copy of a message to multiple bank accounts* - A multicast mechanism is required to allow an ATM to send a single message to all bank accounts simultaneously.
- *Prevent the balance of the replica accounts from deviating, the consequences of which could result in users been presented with bank charges* - Protocols that preserve the total (while still preserving causal) ordering of messages are required.
- *Allowing the audit of a replica account without inhibiting the operation of the other accounts* - A group membership service may identify when an audit is taking place (remove replica) or when an audit is completed (add replica) during the lifetime of a group. In passive replication there is still a need to determine if a member has failed/departed to ensure suitable passive members exist or a failed primary may be replaced.

2.4.4 Summary

The examples described in this section indicate that the existence of a group communication service may benefit the development of distributed services that support groupware and highly available applications. This was achieved by relating the requirements of such applications to properties exhibited by a group communication service. This chapter continues with a detailed description of the mechanisms involved in the provision of a group communication service.

2.5 Group Communications

This section describes the mechanisms commonly associated with a group communication service (originally identified in 2.4.1):

- A multicast mechanism.
- Protocols for managing message delivery.
- A group membership service.

2.5.1 The Multicast Mechanism

To allow a multicast communication each individual member of a group must realise the group membership. This is achieved by allowing each member to maintain a *group view*. By maintaining a

group view a member may identify the address of each group member to which messages may be sent. Each entry in a group view should be an addressable location suitable for enabling the sending of messages to each group member. When a member wishes to multicast a message, the message is sent to every member that appears in the group view.

A desirable property of a multicast mechanism is that a given multicast be reliable (*failure atomic*): if a member crashes while multicasting a message, either all or none of the functioning members deliver the message. Consider the interaction of a client and an active replica group. Multicasts that are not failure atomic may cause problems in maintaining consistency of state between the individual replicas; one replica fails to receive a state-modifying client request but continues to receive and respond to other client requests.

It is sometimes desirable for entities to simultaneously participate in multiple groups [Birman93]. This is certainly true of video conferencing, where users may participate in more than one conference at a time. When an entity belongs to multiple groups a group view for each group must be maintained by the entity. This will enable multicasts to be directed to specific groups.

2.5.2 Messages Ordering

When dealing with a single entity events occur sequentially, each event resulting from some action carried out by the entity. These events are naturally ordered by the sequence in which they happen. A system model may be based on a group of these single entities. Each entity has the ability to send and receive messages to and from other members of the group. The ordering of events in a group is based on two assumptions:

1. The sending of a message m occurs before the receiving of m.
2. If two events occur at the same member then they retain their natural ordering in relation to each other.

A partial ordering of events for distributed systems has been established [Lamport78] based on message passing and the above two assumptions. The notion of "happens before" ($\rightarrow$) is used to indicate partial ordering. The following ordering properties may be derived from previous observations:

- If the event X occurs before the event Y at the same member then $X \rightarrow Y$.
- If the event A is the sending of a message m, and the event B is the receiving of the message m then $A \rightarrow B$.
- If $C \rightarrow D$ and $D \rightarrow E$, then $C \rightarrow E$.

It is possible to state that if $A \rightarrow B$ then A may have caused B. When no causal relationship exists between two events then the ordering between them is arbitrary and two such events may be considered concurrent.

A protocol that introduces ordering highlights a difference between the receiving of a message and the delivery of a message:

- *A* sends the message *m*, *B* receives message *m*, *B* delivers *m*.

Only after a message is delivered may it be accepted by a member for processing. When ordering is relevant a protocol may block the delivery of a message until such a time when ordering requirements are fulfilled. Under certain circumstances a message may never be delivered and so discarded by an order preserving protocol. Following are more detailed descriptions relating to different types of ordering that are common in the support of group communications:

- **Causal Ordering**
  The rules regarding the causal ordering of deliverable messages by a protocol may be derived from the assumption made about causal ordering in an event driven system. As a protocol may concern itself only with the sending and delivery of messages to retain causality it is necessary to block the delivery of a message *m* until all messages that may have caused *m* have been delivered.
- **Total Ordering**
  There are situations in group communications where the delivery of messages to each member should occur in the same order (and preserve causality). Protocols that achieve this are known as *total order protocols*. Protocols that enforce total ordered message delivery must block the delivery of a message until all members of a group mutually agree on the order in which such a message is to be delivered. Total ordering that lacks causal preserving qualities is commonly termed *identical ordering* [Macedo95].

There are two distinct types of protocol for achieving total ordering:

- *Asymmetric* - A single member of the group is responsible for determining the order of delivery.
- *Symmetric* - All members of the group share the responsibility for determining ordering.

In an asymmetric protocol the member responsible for ordering is commonly termed the *sequencer*. Each member of a group may unicast the message they wished distributed throughout the membership of the group to the sequencer. The sequencer is responsible for multicasting such

messages to all members of the group. In a symmetric protocol, members simply multicast their messages to the whole membership of the group.

The asymmetric protocol tends to favor groups that only have a subset of the membership regularly multicasting. Such scenarios arise when clients request a service from a group (as in highly available applications). In a symmetric protocol, ensuring client requests may be suitably ordered for delivery requires all members to participate in a message passing round. This message passing round has to be prompted (on the receiving of a client request) and such message passing may solely exist to order client requests (no computational value to the application). However, if an asymmetric protocol is used members receive client requests already totally ordered and may deliver such messages without the need for further message passing. When all members frequently multicast in a group the symmetric protocol is favored. Such scenarios arise in Groupware applications. In Groupware applications members wish to share information (such as a video image in teleconferencing). Members tend to multicast in an asynchronous fashion (do not wait for reply). As every member frequently multicasts, message passing rounds will be completed. There is no need to prompt message passing solely for the purpose of message ordering. Furthermore, the redirection of messages through a sequencer (as in the asymmetric approach) adds unnecessary message latency in Groupware applications.

2.5.3 Group membership

It is necessary for all members of a group to have a mutually consistent view of the membership of the group (group views of individual members of a group remain mutually consistent). When members do not agree on group membership actions within a group may lead to inconsistencies between the functionality of the group and the group's expected behavior as identified in a specification. For example, consider a simple system that consists of a group of three members (*A*, *B*, and *C*) that service client requests. The following 5 points identify the group's specification:

1. *B* and *C* are backup members for *A* (the *primary* member).
2. Only the primary member may service client requests.
3. When *A* fails *B* should become the primary.
4. When *A* and *B* fail *C* should become the primary.
5. There should always be one, and only one, primary in operation at any one period in time.

An inability to satisfactorily determine mutually consistent group views for each member may result in either one of the following faulty scenarios:

- *No primary exists* - Assume *A* fails. However, *B* does not register this and still includes *A* in its group view. *B* fails to take up the responsibility of becoming the primary, and as *C* does not assume *B* to have failed does not take on the role of primary.
- *Multiple primaries exist* - Assume *C* incorrectly suspects *A* and *B* to have failed, reducing its group view to only include itself. This may result in two primaries, *A* and *C*.

As groups are dynamic (members may join or leave a group), a mechanism that enables some form of consensus on group membership is necessary. The difficulties encountered when determining group membership is referred to as the *Group Membership Problem* (GMP), sometimes referred to as the *consensus problem.*

To aid in solving GMP a *failure detection mechanism* is required to indicate the event of member failure to non-faulty members. This will then enable non-faulty members to install a new group view, excluding any failed members. To enable failure detection it is first necessary to establish the correctness of a member.

2.5.4 Models of Faulty Behavior

Presented here is a classification for expressing faulty behavior of members; this material is taken from [Shrivastava91b, Cristian91a, Ezhilchelvan86].

Assume that the response of a member for a given input is considered to be correct if the output value is as expected and produced on time. This assumption then allows the characterization of failures with respect to both the value and time:

Let a member receive at time *ti* an input requiring a response from the member and as a result produce an output value *v* at time *tj*. For that input, the response *v* at time *tj* is correct iff:

i. The value is as expected: $v = w$, where w is the expected value consistent with the specification; and,

ii. it is produced on time: $tmin \leq tj - ti \leq tmax$, where , $[ti + tmin, ti + tmax]$ is the interval during which the specified output is expected to be produced.

The values *tmin* and *tmax* are constants of a given member. First of all, note that the response of a member cannot be instantaneous to a given input but must experience a finite minimum amount of delay (which is specified by the parameter *tmin*). The maximum delay time, *tmax* indicates the upper bound on the output delay.

A correctly functioning member does not arbitrarily produce responses. In particular, when there is no input (null input) or when no response is expected for an input, naturally no output value is

produced (output is null). The values *tmin* and *tmax* are meaningful only when a member is expected to produce a response.

If $v \neq w$, then the *output value will be termed incorrect*; similarly, if $t_j < t_i + t_{min}$ (output produced too early) or $t_j > t_i + t_{max}$ (output produced too late), then the *response time will be termed incorrect.*

Given the above definitions of correct and incorrect responses, there can be four possible ways by which a response can deviate from that specified. This leads to the following types of faults.

1. *Timing Fault* - A fault that causes a member to produce the expected value for a given input either too early or too late will be termed a timing fault and the corresponding failure a timing failure.

   $v = w$ **and** $(t_j - t_i < t_{min}$ **or** $t_j - t_i > t_{max})$.

   A late timing fault is also referred to as a performance fault [Cristian91a].
2. *Value Fault* - A fault that causes a member to respond, for a given input, within the specified time interval, but with a wrong value will be termed a value fault and the corresponding failure a value failure:

   $v \neq w$ **and** $t_{min} \leq t_j - t_i \leq t_{max}$.

3. *Omission Fault* - A fault which causes a member, for a given input requiring a non-null response, not to produce a response will be termed an omission fault and the corresponding failure an omission failure. The act of 'not producing a response' is regarded as equivalent to 'producing a null value on time', thereby treating an omission fault as a special case of a value fault. An omission fault can also be treated as a special case of a timing fault by regarding 'not producing a response' as equivalent to 'producing a correct value at infinite time'.
4. *Byzantine Fault* - A Byzantine fault is responsible for a Byzantine (*fail-uncontrolled*) failure which is any violation from the specified behavior. In particular, it includes the possibility of a member producing a response when no input was supplied. A Byzantine member is customarily considered in the literature to be capable of being 'malicious' in its responses [Lamport82].

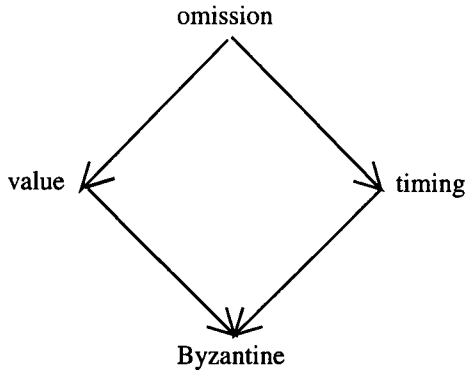

Figure 2.7 - Fault Lattice.

A Byzantine fault (failure) subsumes all the other three types of faults (failures). The relationships among these four types of faults (failures) can be expressed by the fault (failure) lattice shown in fig. 2.7, where an arrow from A to B, indicates that fault (failure) type A is a special case of fault (failure) type B. (The relation '$\rightarrow$' is transitive.) An important observation can now be made which is that a fault-tolerant algorithm designed to tolerate m, m > 0, timing failures (value failures) can also tolerate m omission failures and further that an algorithm designed to tolerate m Byzantine failures can tolerate m failures of any type. The top of the lattice represents the fault type with most restrictions and the bottom with the least.

The above classification is based on the behavior of a component with respect to a single output. When a sequence of outputs over a given time interval is considered, the type of fault in the component will be taken to be the least restrictive one of which all types of failures occurred during that interval can be considered to be special cases. If a given faulty behavior persists for a 'sufficiently' long time, then that failure can be considered to be permanent. In particular, a component that suffers a persistent omission failure will be said to have failed in a *fail-silent* manner (also referred to as a *crash-failure*). Non persistent failures are called *transient* failures. In the Thesis, members will be assumed to be fail-silent. This is the assumption made in all other group communication services (some of them will be reviewed later in this chapter).

2.5.5 Group Membership and Virtual Synchrony

The nature of the distributed computer system environment is an important consideration when solving GMP. There are two basic ways of modeling computer system environments:

- *Synchronous* - Process and network delays are known and bounded.
- *Asynchronous* - Process and network delays are bounded but unknown.

As the Thesis is concerned with group communication protocols for Middleware, an asynchronous environment is assumed. This is because Middleware is commonly presented over computing systems which are heterogeneous in nature and communicating via the Internet. Ensuring synchronous assumptions hold in such an environment is unrealistic.

The asynchronous environment assumption has a strong implication on failure detection, since under the circumstances it is impossible to distinguish between a member that has suffered failure (or disconnected from the network) and a member that is simply slow. If a member does suffer from failure, it is important that non-faulty members forming a group come to some agreement on that failure event and remove the failed member from the group; this is impossible to achieve in finite time [Fischer85]. To circumvent this impossibility result group communication protocols permit members to be *suspected* of failure [Chandra91, Schiper93]. This enables members of a group to mutually agree on the membership of a group. Of course, there is the possibility of a correctly functioning member to be erroneously removed from a group [Chandra91].

Group membership and protocols that ensure ordered atomic message delivery are combined within a group to enable the *virtual synchrony model* [Birman87]. The virtual synchrony model orders group membership changes with respect to computational messages multicast within a group. It ensures that view changes are "seen" identically by members. Furthermore, the virtual synchrony model ensures that two members that proceed together from one view of group membership to the next deliver the same messages in the first view.

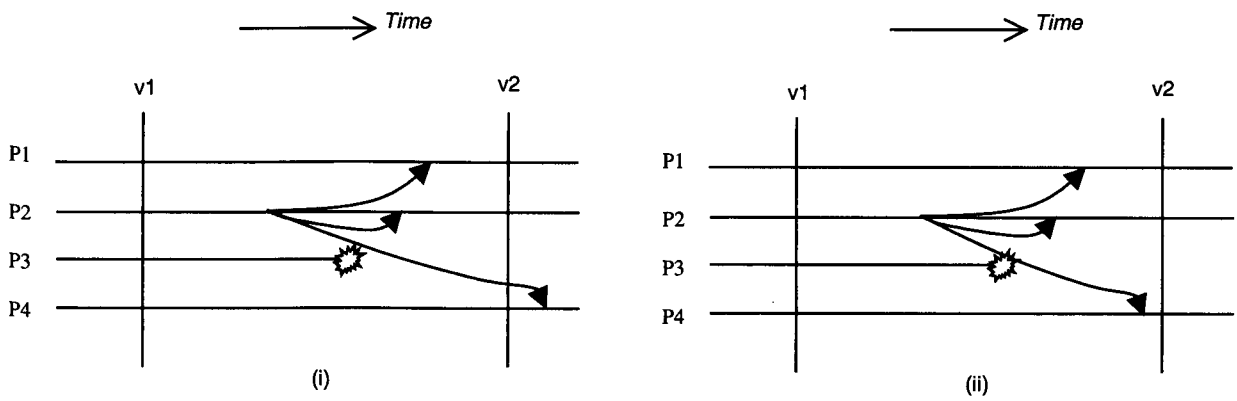

Figure 2.8 - Virtual synchrony.

The diagram in fig. 2.8.i illustrates non-virtual synchronous communications. The multicast of *p2* is delivered in view *v1* by *p1* and *p2*, whereas *p4* delivers the multicast of *p2* in *v2*; members that proceed together from one view of group membership to the next are not delivering the same

messages in the first view. The diagram in fig. 2.8.ii does illustrate virtual synchronous communications. The multicast of *p2* is delivered by all members in the same view; members that proceed together from one view of group membership to the next are delivering the same messages in the first view.

A partitioning of a group is reflected in the group views of the members. For example, in fig. 2.9.i *D* and *E* have failed resulting in *A*, *B*, and *C*, removing *D* and *E* from their group views.

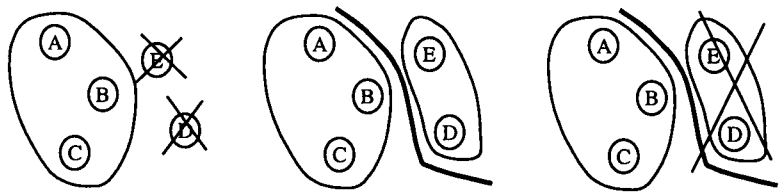

Figure 2.9 - The partitioning of groups.

After a network partition has occurred more than one group may exist. In our example *A*, *B*, and *C* may suspect *D* and *E* to have failed, and so exclude them from the group during a group view update. However, a network partition may have occurred, leaving two groups (fig 2.9.ii). In this scenario, *A*, *B*, and *C* will have excluded *D* and *E* from their group views while *D* and *E* will have excluded *A*, *B*, and *C* from their group views. Due to the ability of a failure detector to suspect non-faulty members it is also possible for a subgroup of members to wrongly agree on one or more correctly functioning and connected processes that may still be participating in the group as having failed. This will lead to groups partitioning themselves with no failures present (a *virtual partition*).

In a primary partition membership service only a single group (usually the one with the majority of members) is maintained (fig. 2.9.iii), whereas in a partitionable membership service, all the subgroups survive (fig. 2.9.ii). At a later time these subgroups may be merged.

This chapter continues by describing the various ways the group communication mechanisms detailed in this section may be introduced to CORBA.

2.6 Introducing Group Communications to CORBA

Providing group communication support for use in the CORBA has been approached in three ways [Felber98b]:

- *Integration* - A group communication service has been integrated into an ORB.
- *Interception* - Messages issued by an ORB are intercepted and mapped onto an underlying group communication service.

- *Service* - Group communications are supplied as an object service.

These approaches are now described in more detail.

2.6.1 Integration Approach

To enable group communications an existing group communication service is imbedded within the structure of an ORB. Object groups are managed within the ORB by an application via extensions to the standard interfaces presented by the ORB BOA interface or via information stored in information stores commonly used to identify information relating to services (commonly termed *repositories*).

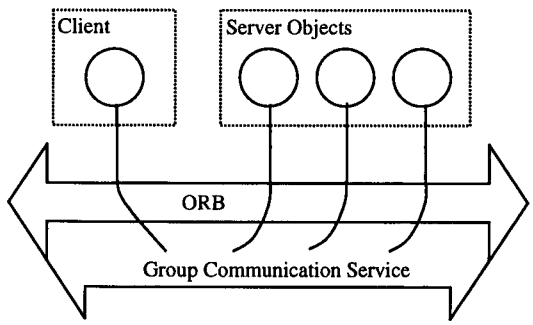

Figure 2.10 - The integration approach to CORBA group communications.

Group transparency is associated to client requests; the client views invocations directed at a group in the same manner as those directed at a singleton. For this reason only one reply is returned to the client per request. To return more replies requires an enhancement to the IDL definition of a service. This enhancement takes the form of changing return values to a list of the same return values, enabling each server's response to be placed in an element of the list. Lists are adequately accommodated for in IDL by the *sequence* data structure, a data structure that may be of variable length and hold data items of arbitrary type. This has resulted in the need for two types of proxy code to be produced to satisfy a clients needs. The standard type that returns values as if a group is a singleton or the less transparent type that is capable of returning a number of server replies.

2.6.2 Interception Approach

The interception approach requires no alteration to the infrastructure of the ORB to enable object groups. The messages sent between objects are intercepted and, if group communications are required, mapped onto an underlying group communications service.

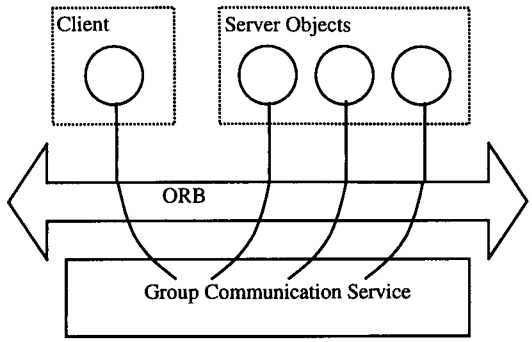

Figure 2.11 - The interception approach to CORBA group communications.

To retain the interoperability qualities of the ORB, the enhancement of the BOA to provide group management functionality is not an option. Therefore, services for managing object groups (member departures/joins, group creation, group deletion) are presented to the application developer either as object services or via a management service interface directly exposing the underlying group communication service to applications. In effect, even when the object service approach is used functionality is still mapped to the underlying group communication service.

Gaining multiple replies (reply from each group member) from a single client request requires the type of IDL enhancements of those mentioned in the integration approach.

2.6.3 Service Approach

The service approach adds a group communication service as an object service. The object service approach is recognized by the OMG as the standard method for enhancing ORB functionality and is therefore more in line with CORBA application development. However, to ensure the interoperability of an object service, only the standard ORB message passing mechanisms are available (point-to-point RPC) for enabling multicasts. Therefore, a multicast actually consists of a number of RPCs, each one directed to a group member. A problem with this method of enabling multicasts is the synchronous nature of the communications supported by an ORB; a method invocation to a remote object blocks the calling object's processing until the request has been handled by the server object (this happens even when no reply is expected). As a message ordering protocol may block message delivery, or even discard a message, the availability of an asynchronous message passing environment is essential to ensure the progress of an application's execution during any group communication activity; a client that makes an RPC may continue processing as the RPC is handled by the server. There are two methods to enable this:

- *OneWay Call* - The oneway call represents an asynchronous message passing mechanism within CORBA. Unfortunately, the specification describing the oneway call indicates that the use of

oneway does not guarantee message delivery. Some ORBs may never deliver a oneway call, whereas others will always deliver a oneway call. The various ways vendors have interpreted and implemented the oneway call over a number of ORBs makes the oneway call unsuitable for distributed application development; the requirements placed on the oneway call by an object may not be satisfied on some ORBs.

- *Thread* - When an asynchronous RPC is required a process spawns a thread which handles the RPC.

The choice between thread and oneway depends on application requirements and the ORB been used. However, the thread approach ensures the correct working of an object service irrelevant of the vendor that supplied the ORB.

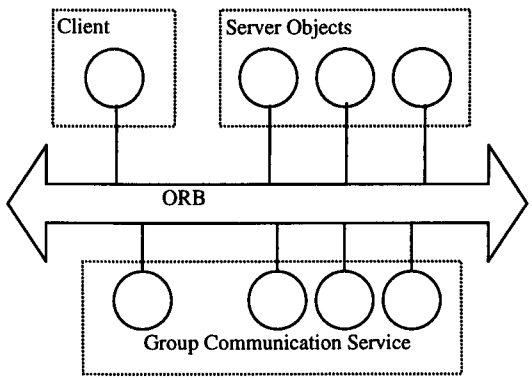

Figure 2.12 - The service approach to CORBA group communications.

A group communication service may be implemented as one or more services (CORBA objects) that implement all the required mechanisms to satisfy an application's group communication requirements.

2.6.4 Evaluation

The choice of approach when providing group communications via the methods expressed in this section effect the following two properties of an application:

1. *Interoperability* - The ability of a service to work with any ORB, irrelevant of underlying network services and operating system.
2. *Performance* - The message latency incurred when using group communication mechanisms.

Of the three approaches the service approach has the potential to incur the greatest performance overhead. This is because each multicast is split into a series of RPCs, each RPC passing through the ORB infrastructure and therefore incurring overheads related to time taken up by IIOP and ORB dependent mechanisms (e.g., marshaling and unmarshaling). This may not be too drastic a problem, but all protocol related messages, not just application related, must pass through this expensive route. Such messages may be required to force message delivery or agree on group membership. The integration and interception approach, however, restricts only application derived request/reply messages to passage through an ORB. All other messages, related to the management of the group communication protocols themselves, are restricted to transport level protocols. This allows the integration and interception approaches to make use of existing network technologies that significantly increase the performance of group communications (e.g., IP multicast, LAN broadcast). It is expected that CORBA will be extended with multicast facilities to enable exploitation of transport level multicast/broadcast facilities. Then the service approach will no longer be associated with poor performance.

The benefit of the service approach is its ability to preserve the interoperability qualities currently enjoyed by applications developed in the CORBA environment. The integration approach relies on a non-standard extension to the ORB interfaces to enable object groups. This makes applications dependent on a particular vendor's ORB. The interoperability aspects of the interception approach are not as severe as the integration approach, however, reliance is placed on underlying network and operating system services.

Making a request directed at an object group appear the same as a request directed at a singleton (group transparency) is slightly more difficult in the service approach. This is because, unlike the integration and interception approach, there is no scope for presenting an IOR that represents a group. The existence of such an IOR is not possible in CORBA. However, there is an ability to present an IOR that conforms to the format as identified by the CORBA specification but has additional information imbedded within it to enable group communication services that are integrated or interception based to associate an object group to such an IOR. The objects in the CORBA environment are thus presented with group IORs that appear indistinguishable from those IORs that belong to a singleton. To achieve the same transparence in the service approach requires the use of a proxy object. A single object with an IOR that presents a single interface representing the functionality of the group. Clients direct requests to the proxy object, the proxy object then forwards requests and receives replies, returning replies back to the client.

2.7 Related Work

I shall now describe existing implementations that attempt to provide group communication mechanisms for use by application developers in CORBA.

2.7.1 Orbix+Isis

Orbix+Isis [Iona94] is an example of the integration approach. An existing group communication service (Isis [Birman93]) has been integrated into an ORB (Orbix [Iona94]). The description of Orbix+Isis continues with a brief overview of the Isis system itself.

Isis provides the low-level communication abstractions required to build fault-tolerant distributed systems. A reliable multicast service with multiple ordering primitives is supplied. The CBCAST protocol guarantees causal order, while ABCAST is a total order protocol built on top of CBCAST. Causal multicast over multiple overlapping groups is provided by an extension of the CBCAST protocol.

By integrating the multicast service with a membership protocol Isis provides virtual synchronous process group communication (the first system to achieve this). When a group is partitioned (be it network or virtual), the continuation of group services are undertaken by the primary partition (usually the partition with the most members), all former group members that are not members of the primary partition are killed off. Note that circumstances may arise when no distinct group may attain the status of primary, indicating the failure of the group.

Multicast based communications in Orbix+Isis are handled by Isis. Point-to-point communications are handled by Orbix. Orbix+Isis aims to provide highly available services via replica object groups. The virtual synchronous, total ordered message delivery protocols provided by Isis enable object groups to remain mutually consistent. When communicating with an object group in a synchronous manner a client may enable one of the following communication mechanisms:

- *Choice* - Invoke method of a single member only. Due to the possibility of state deviation this should be limited to read-only accesses.
- *Multicast* - Invoke method at all objects in a group.
- *Coordinator/cohort* - Invoke method of a single member only. However, this member is then responsible for indicating to the other members of the group any changes in state that may have occurred. This ensures that states remain mutually consistent.

Support exists for asynchronous client-to-group communications. This support resembles the CORBA Event Service. Clients place messages on what Orbix+Isis terms an *event stream*, servers take messages from an event stream. The major difference between the event stream and the event service is that the event stream is actually replicated, enabling the event stream to be highly available.

Information that govern client/group interaction and group management issues are dictated by information stored in two repositories:

- *Implementation repository* - Enabling the correct directing of client requests to suitable CORBA objects.
- *Isis Repository* - Indicates the group management aspects of a group, such as the number of replies to be returned and the communication mechanism to use.

2.7.2 Electra

Electra [Maffeis95] is another example of the integration approach. Electra describes the complete programming environment, including the ORB. Rather than combining existing group communication technologies with an existing ORB, the developers of Electra built the ORB complete with group communications as an integral part of the ORB's design. This has allowed the Electra environment to be tailored to make use of a variety of existing group communication services, such as Isis. The well known implementation of Electra (as described by its author in his Ph.D. Thesis [Maffeis95]) utilizes the Horus group communication service. For completeness, a brief overview of Horus is now given.

Based on the lessons learned when using the Isis system, the Isis research team developed Horus [Renesse96]. Horus provides a flexible group communication model with the ability to support virtual synchrony when required. Protocols that support group communications may be created by stacking protocol modules that have uniform interfaces. Each protocol module has a separate responsibility. For example, a module located near the top of the stack may be responsible for ordering guarantees relating to message delivery, whereas a module located at the base of the stack may be responsible for accessing the appropriate interface to the underlying network. The structured framework for protocol composition incorporates ideas from such systems as the UNIX "streams" framework [Ritchie84] and the X-Kernel [O'Malley89]

There is an ability to tailor Horus to suit a wide variety of application types. If the stacking of existing protocol modules does not suit an application's requirements new protocol modules may be created by a developer. An additional feature of the Horus system is its ability to extend the virtual synchrony model of Isis to reduce the need for message blocking. This new virtual synchrony model is termed *weak virtual synchrony*. In the weak virtual synchrony model messages are still guaranteed to be delivered in the view they were sent, however, the notion of which view a message was actually sent in is slightly weaker. The membership protocol of Transis [Kramer91] (that allows partitionable operation) is the basis for the membership protocol of Horus.

The ability to configure transport level protocols to suit a wide variety of application types is one of the popular features of Horus. However, the stacking of protocols is assumed to have been accomplished before use with Electra may commence. Electra does not provide high level interfaces that allow the configuring of protocol stacks.

Electra extends the BOA interface with mechanisms suitable for group management. As all objects have access to the BOA interface, any object may participate in group communications. However, supplying these mechanisms to all objects may be viewed as wasteful, as not all objects will participate in group communications.

From an IDL interface that defines a service supported by an object group, Electra may generate two versions for each method present. One method, the standard version, enables group transparency whereas the second method, enhanced version, enables a client to receive replies from all members of a group.

2.7.3 Eternal

The Eternal system [Narasimhan97] is an example of the interception approach. Invocations that objects make on each other via the IIOP are intercepted by the Eternal system. When group communication is required, Eternal maps these invocations onto a reliable, total order, group communication service. The underlying group communication service used by Eternal is Totem [Agarwal94].

Totem provides reliable, totally ordered multicasting of messages over LANs and exploits the broadcast facility of such networks to achieve high performance. The replication of information to improve availability was an important consideration in the development of Totem. The total ordering of messages allows the consistency of such information to be achieved.

Totem manages group view updates and message delivery in accordance to the extended virtual synchrony model [Melliar-Smith94]. Unlike the virtual synchrony model [Birman93], the extended virtual synchrony model allows partitionable operation and guarantees that messages are delivered in a unique order within disconnected (partitioned) components. Achieving faster message delivery by utilizing network multicast technologies within a virtual synchronous type group communication environment for partitionable group operation are the main goals of the Totem research efforts.

Fig. 2.13 shows an architectural overview of the Eternal system. The four parts that constitute the Eternal system are:

- *Fault Tolerance Management Services* - Enable application developers to describe the high availability properties of an application.
- *Fault Tolerance Core Services* - Enable an application program to exercise dynamic control over high availability mechanisms (such as replication of service and recovery from failures).
- *Fault Tolerance Mechanisms* - Provides an efficient implementation of the infrastructure required by the previous two services.
- *Multicast Engine*- The multicast engine provides group communications via Totem.

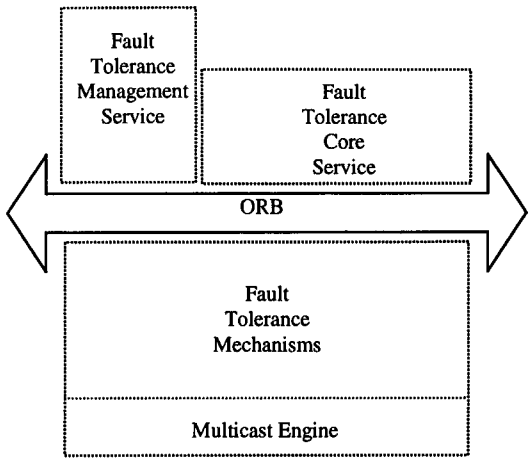

Figure 2.13 - Architecture of the Eternal system.

The services are implemented as object services and provide application developers with interfaces to the lower level fault tolerance mechanisms.

2.7.4 Object Group Service

The Object Group Service (OGS) [Felber98a, Felber98b] Provides group communications as a CORBA service. This ensures that the interoperability and portability associated with CORBA are preserved: the ORB requires no special enhancement (as in the integration approach) and there is no need for operating system and network service dependent communication mechanisms (as in the interception approach).

Mechanisms are provided that allow an application developer to choose between arbitrary, causal or total ordered message delivery within a group. Further mechanisms are supported to enable the management of replica groups (e.g., state transfers), and to allow two distinct flavors of replication protocols (active and passive) to be used on a per message basis. Mechanisms are presented to the application developer via the following services:

- *Messaging Service* - Provides reliable asynchronous point-to-point and multicast communications.
- *Monitor Service* - Monitors objects and acts as a failure detector.
- *Consensus Service* - Enable protocols for managing multicasts and group membership.
- *Group Service* - Provide group multicasts and group membership.

A client may determine the type of replies expected from a group; a wait for all replies, wait for majority of replies, wait for first reply. OGS allows clients to interact with object groups via the sending of *untyped messages* or *typed messages*. An untyped message is a message that is passed to OGS for multicasting to an object group. The client is aware that a multicast is occurring, and is responsible for marshaling the parameters of a call into a suitable data structure to allow OGS to enable the multicast. The data type used is the CORBA type *any*. The any data type is capable of holding any CORBA data type, including sequences, and is therefore ideal for use in marshaling. This method of interacting with a group allows a client to trivially identify the number of replies to be expected from the group (e.g., wait for all, wait for majority). The drawback to this method is the problems associated with marshaling data, an error prone task, and the exposure of the group communications to the client (non-transparent groups). Clients issuing typed messages assume that they are directing messages at a singleton, as opposed to a group (group transparency). These messages actually arrive at OGS, which performs the multicast, filters replies, and returns a single reply to the client. This is achieved with the aid of the DSI and DII. The OGS gains knowledge of a service interface at runtime via the *interface repository*, uses the DSI to enable client requests directed at this service to be handled by OGS. OGS then uses the DII to actually perform the multiple RPCs that make up a multicast on the server objects. Finally, a single reply is returned to the client

2.8 Summary and Contribution Made by Thesis

This chapter concentrated on a number of existing Middleware technologies for enabling distributed application development in an object-oriented style. The three most popular Middleware technologies (Java-RMI, DCOM, CORBA) were briefly described, followed by a more in-depth description of CORBA. The lack of support for object groups in the CORBA (and also Java-RMI and DCOM) standard was highlighted, followed by descriptions of two types of applications (highly available, Groupware) that may benefit from such support. Services (multicast, message ordering, atomic message delivery, membership) that enable group communications were then described in more detail followed by the three methods (integration, interception, service) that may be used to incorporate such services into CORBA. Finally, works related to this area of research were presented that exemplify the three approaches to incorporating group communications into CORBA.

As can be seen from this chapter, research into group communications for Middleware environments has resulted in a number of services that enable an application developer to make use of group communication protocols in their applications. However not every service supports all the functionality that may be required. This reduces application developers to a choice that will "best fit" an application's requirements.

Evaluation of the application requirements and the functionality provided by existing group communication services surveyed in this chapter has resulted in the following list of items that, if configurable by an application developer, may ease the suitability of a group communication service for an application's requirements:

- *Ordering and reliability guarantees* – Causally or totally ordered atomic message delivery.
- *Protocol used to enable message ordering* – Symmetric or asymmetric.
- *Overlapping groups* - Group members may participate in multiple groups simultaneously.
- *Per group ordering protocols* - Multi-group members may participate in an asymmetric ordering protocol in one group while participating in a symmetric ordering protocol in another group.
- *Protocol timeouts used for failure suspicion* – set on a per node, member, or group basis.
- *The existence of groups in the presence of network partitions* – Partitionable operation or insist on a primary partition.
- *Flexible client/server interaction* - A client does not have to multicast to the full membership of a group in order to access services provided by a group. This is particularly important in an Internet environment, as discussed below.
- *Middleware service* - The increased functionality that is required to support group communications should be provided in way that does not inhibit the interoperability an application would otherwise enjoy in a Middleware environment.

The group communication services surveyed in this chapter concentrate only on presenting an environment suitable for object replication. Such environments ensure, in part, replicas remain mutually consistent by totally ordering message delivery within a group. OGS allows an application developer to choose between total and causal ordering. Other approaches (integration and interception) shield the application developer from protocol details and so inhibit any manipulation of them. Furthermore, no existing system allows the application developer to choose the type of protocol used to order messages; an application developer may not choose between symmetric or asymmetric message ordering protocols. There are group communication protocols that enable objects to participate in multiple groups, however, to the best of our knowledge, there are no group communication protocols that allow a multi-group member to use, say, asymmetric ordering in one group while using symmetric ordering in another group. As mentioned earlier in this chapter (see 2.5.2), asymmetric protocols are more suited to replica service provision whereas symmetric protocols are suited to Groupware applications.

Existing group communication protocols presented as Middleware services enable client access of server groups by regarding clients as full and/or special members of the server group. The term "special member" indicates a client that communicates with the server group by multicasting to

the full membership of the server group, however, the client does not appear in the group views of the server group nor does it participate in group membership agreement with the server group. In this scenario, server group members must decide upon the ordering and delivery guarantees of client requests via extra message passing or network related timestamps associated to the requests [Agarwal94]. This type of client/server group interaction is sometimes termed *closed.* There are instances when a client may wish to access the services provided by a group by sending a request as a unicast. For example, when a client is geographically separated from a server group (such as a client accessing services over the Internet) sending a message to each member of the server group will be substantially more time consuming than simply sending a single message to a single server due to the large message latency experienced on a WAN (e.g., Internet). This method of client/server group interaction is commonly termed *open.*

Services that rely on non-standard mechanisms to be present in the Middleware environment inhibit the interoperability of applications built using them. Orbix+Isis and Electra extend the functionality of an ORB making applications built using them ORB dependent; ORBs supported by other vendors will not be able to support applications built utilizing the group communication mechanisms found in these tools. The Eternal system does not extend the functionality by altering the infrastructure of the ORB, but does use an interception method that is operating system dependent (UNIX) and relies on the existence of a group communication sub-system (Totem) that depends on the existence of a network broadcast facility that is limited to a LAN and is not suitable for use in a WAN. This, again, reduces interoperability; ORBs that do not have access to such an operating system and/or support applications that are distributed over a WAN may not make use of the Eternal group communication service. The OGS does maximize interoperability as all multicast operations are carried out using CORBA compliant point-to-point communications (i.e., via the ORB). There is no reliance on any group communication sub-system or non-standard protocols that enhance the ORB. However, OGS does not provide support for open groups nor does it allow overlapping groups.

In summary, existing group communication services do not support all of the requirements stated earlier. The Thesis presents the results of research into the provision of a CORBA object group service that is suitable for a wide variety of application types as it supports all the requirements stated earlier. Due to the interoperability qualities, the OGS approach is adopted by the Thesis to enable the provision of group communications for use in CORBA. Furthermore, protocols are presented that enable open and closed client/server group interaction. This service is called the NewTOP service. In chapter 3 protocols that enable fault-tolerant group communication within the service are described. These protocols were developed earlier at Newcastle University [Macedo95, Ezhilchelvan95]. Chapter 4 describes how these protocols were implemented as a CORBA service. Chapter 5 describes invocation protocols for allowing clients to interact with server groups via open and closed methods.

The performance of the NewTOP service is presented in chapter 6. Chapters 4, 5 and 6 represent the main contribution of the Thesis. Finally, chapter 7 summarizes the Thesis.

Chapter 3

The NewTOP Group Communication Protocols

In this chapter the protocols that are suitable for implementing a Middleware service that may provide group communications for a wide variety of application types are described. The initial work related to the design of the protocols described in this chapter may be found in [Macedo95] and [Ezhilchelvan95]. These works describe the NewTOP (Newcastle Total Ordered Protocol) protocol suite. NewTOP provides reliable causality preserving total order message delivery to members of a group, ensuring that total order message delivery is preserved for multi-group members. The initial sections of this chapter describe the basic concepts and provides descriptions of the protocols in a static group (membership remains constant) scenario. This constraint is lifted and the mechanisms required to ensure the correctness of the protocols when group membership may be dynamic are then described. Mechanisms related to the optimization of these protocols in the presence of multi-group members are then presented. Finally a summary of the work presented in this chapter is provided, comparing the protocols presented here with the equivalent protocols developed for use in similar works.

3.1 Basic Concepts

A group is defined as a collection of distributed entities (process, object or module) in which a member communicates with other members only by multicasting to the full membership of the group. A given entity can be a member of more than one group. A member execution consists of a sequence of events, each event corresponding to the execution of an action by a member. $send_i(m)$, $receive_i(m)$ and $delivery_i(m)$ will be used to denote the events of sending, receiving and delivering a message m by a member P_i respectively. (The suffix i may be dropped if the identity of the member executing the action is not important).

An assumption is made that the underlying network supports uncorrupted and sequenced message transmission between sender and destination (first in first out, FIFO, message transmission), if the sender and destination are functioning correctly and the destination is not partitioned from the sender. Communication failures could lead to network partitions causing the members of a group to be split into disjoint sub-groups, with the functioning members in one subgroup unable to communicate with the functioning members in the other sub-groups. As an asynchronous communication environment is assumed (see 2.5.5), no assumption about message transmission time will be made.

Let G_i be the set of groups P_i belongs to: $G_i = \{g_x \mid P_i \in g_x\}$. The membership of P_i in a given group will be denoted as g_x, $g_x \in G_i$ and let $g_x = \{P_1, P_2, \ldots P_n\}$. When P_i multicasts (or delivers) a message m with $m.g = g_x$, it actually does so only to (or from) those entities which it *views* as functioning members of g_x. P_i delivers its own messages also by executing the protocol in operation. When g_x is initially formed, each functioning P_i installs an initial view $V^0_{x,i}$, say, $V^0_{x,i} = \{P_1, P_2, \ldots P_n\}$. If P_i is unable to communicate with some $P_k \in V^0_{x,i}$ (this could be because P_k has failed or disconnected or departed from g_x), it installs a new view that does not include P_k. Let $V^0_{x,i}$, $V^1_{x,i}$, $V^2_{x,i}$,... $V^r_{x,i}$ be the series of views P_i has sequentially installed over a period of time, until it fails or leaves the group g_x. Note that once P_i leaves g_x, it maintains no membership view for g_x. Each P_i is provided with a *group view* process, denoted as $GV_{x,i}$, for each g_x, $g_x \in G_i$. The group-view process $GV_{x,i}$ makes judicious use of timeouts for suspecting the absence of member processes; it executes a *membership protocol* with other members of the group to reach agreement on these suspicions, which if confirmed lead to an update of membership view (installation of a new view) of P_i for group g_x.

A new view will always be a proper subset of the old view(s) since entities do not join the group they have departed. Entities wishing to join their former co-members do so by forming a new group. An entity can take part in the formation of a new group while retaining its existing memberships. This eliminates the need to support an explicit facility for process joins. Former members creating a *new* group is the equivalent to the former members of a group rejoining the same group with *new* identifiers.

The *service membership* protocol maintains view consistency in the presence of (real or virtual) partitions by permitting a group of entities to partition themselves into two or more sub-groups of connected members with the property that:

1. The functioning members within any given subgroup will have identical views about the membership; and
2. The views of members belonging to different subgroups are guaranteed to stabilize into non-intersecting groups.

When a group partitions into subgroups, members of every subgroup will consider themselves as the sole surviving members of the original (unpartitioned) group, and will not know the existence of other subgroups and their memberships. It is left to applications to decide whether or not to maintain more than one subgroup.

View updates must satisfy certain conditions so that message delivery can be 'atomic' with respect to view updates (see 2.5.5 - group membership and virtual synchrony). Therefore, view updates performed by members of a group g_X satisfy the following *view consistency* (VC) properties:

- **VC1:** The sequence of views installed by any two functioning members of g_x that do not suspect each other are identical *(validity).*
- **VC2:** If a $P_k \in V^r_{x,i}$ leaves g_x or fails or gets disconnected from P_i and if P_i does not fail, then P_i will eventually install $V^{r'}_{x,i}$ such that $r' > r$ and $P_k \notin V^{r'}_{x,i}$ *(liveness).*
- **VC3:** any two functioning members deliver the same set of messages between two consecutive views that are identical. That is, $V^r_{x,i} \equiv V^r_{x,j}$ and $V^{r+1}_{x,i} \equiv V^{r+1}_{x,j} \Rightarrow$ the set of m, $m.g = g_x$, delivered by P_i and P_j in V^r_x is identical.

In the presence of member failures and departures, the following message delivery (MD) properties for all m and m' multicast with $m.g = m'.g = g_X$ (in stating them the suffix x will be dropped when only the group g_x is considered). We will use the notation $m.s$ to denote the sender of m and the symbol $\rightarrow$ will denote "happens before" relationship:

- **MD1** *(validity)*: *for any* m *and* $r \geq 0$: $delivery_i(m,r) \Rightarrow m.s \in V^r_i$. In words: a member will deliver a message m in view V^r, only if the sender of m is in V^r.
- **MD2** *(liveness)*: *for any* m *and* $r' \geq r \geq 0$: $send_i(m,r) \Rightarrow$ *either* $send_i(m,r) \rightarrow delivery_i(m,r')$ *or* $send_i(m,r) \rightarrow$ *failure of* P_i . In words: if a P_i sends m in view V^r_i, then provided it continues to function, it will eventually deliver m in some view $V^{r'}_i$, $r' \geq r$.
- **MD3** *(atomicity)*: $\forall P_i, P_j$ *s.t.* $V^r_i \equiv V^r_j \wedge V^{r+1}_i \equiv V^{r+1}_j : delivery_i(m,r) \Leftrightarrow delivery_j(m,r)$. This property is implied by VC3.

Properties MD1 to MD3 together ensure live, atomic delivery in the presence of dynamic membership changes. The additional property MD4 (and its extension for multiple groups, MD4') ensure causality preserving total order message deliveries:

- **MD4** *(total order, single group):* $\forall P_i, P_j$ *s.t.* $V^r{}_i \equiv V^r{}_j \wedge V^{r+1}{}_i \equiv V^{r+1}{}_j$: $delivery_i(m,r) \rightarrow delivery_i(m',r) \Leftrightarrow delivery_j(m,r) \rightarrow delivery_j(m',r)$; if $delivery_i(m,r)$ and $delivery_i(m',r')$ occur for a given P_i then $m \rightarrow m' \Rightarrow delivery_i(m,r) \rightarrow delivery_i(m',r')$.

The above delivery order is extended for messages multicast in different groups, ensuring a total delivery order when the same messages are delivered to entities that simultaneously belong to multiple groups. Let μ be a message with $\mu.g = g_y$ and $\rho \geq 0$ be an integer:

- **MD4'** *(total order, multiple groups):* $\forall P_i, P_j$ *s.t.* $V^r{}_{x,i} \equiv V^r{}_{x,j} \wedge V^{r+1}{}_{x,i} \equiv V^{r+1}{}_{x,j} \wedge V^{\rho}{}_{y,i} \equiv V^{\rho}{}_{y,j} \wedge V^{\rho+1}{}_{y,i} \equiv V^{\rho+1}{}_{y,j}$: $delivery_i(m,r) \rightarrow delivery_i(\mu,\rho) \Leftrightarrow delivery_j(m,r) \rightarrow delivery_j(\mu,\rho)$; if $delivery_i(m,r)$ and $delivery_i(\mu,\rho)$ occur for a given P_i then $m \rightarrow \mu \Rightarrow delivery_i(m,r) \rightarrow delivery_i(\mu,\rho)$.

For a given delivered m', MD5 states the situations in which the delivery of a causally precedent m, $m \rightarrow m'$, is guaranteed:

- **MD5** *(causal prefix):* *for any m and m' s.t.* $m \rightarrow m'$: $delivery_i(m',r') \Rightarrow delivery_i(m, r)$. In words: if m' is delivered to P_i in view $V^r{}_i$ then every m, $m \rightarrow m'$ *and* $m.g=m'.g$, is delivered to P_i in some view $V^r{}_i$. (Note that MD4 implies that $delivery_i(m,r) \rightarrow delivery_i(m',r')$.). MD5 is also respected in overlapping groups (see [Ezhilchelvan95]).

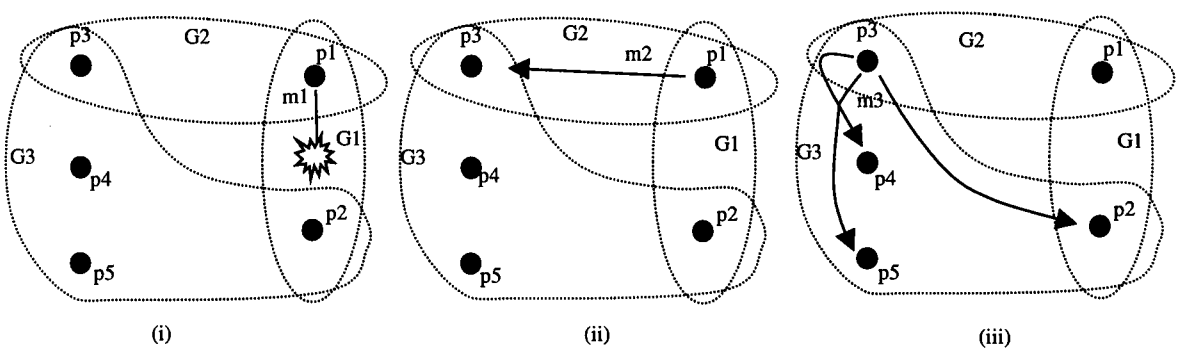

Figure 3.1 - Total order message delivery in overlapping groups.

The diagrams in fig. 3.1 aid in the understanding of causality preserving total ordered message delivery for overlapping groups. Message *m1* is multicast in group *G1* by *p1*. However, before *m1* can be received by *p2* a network partition occurs (fig. 3.1.i). *p1* then multicasts a message (*m2*) in group *G2* (fig. 3.1.ii). *p3* receives message *m2* and then proceeds to multicast a message *m3* in group *G3* (fig. 3.1.iii). There exists a causal chain of messages such that $m1 \rightarrow m2 \rightarrow m3$. Members that receive these messages should delivery them in an order that respects the causal relationship between them. Therefore, *p2* can only deliver *m3* if one of the two following scenarios can be satisfied:

- *m1* is somehow retrieved and delivered to *p2* before *m3* is delivered by *p2*.
- If *m1* cannot be retrieved by *p2* before *m3* is delivered by *p2* then *p1* should be excluded from *p2*'s view for *G1* before *m3* is delivered by *p2*.

In the latter case, the network failure that occurred during the multicast of *m1* is perceived by *p2* to have happened before the multicast of *m*1; the total ordering of events by *p2* would indicate that *p1* was excluded from *p2*'s view of *G1* before *m1* was multicast.

3.2 Ordering Protocols

As mentioned in the last chapter, message ordering may be achieved via asymmetric or symmetric protocols. These protocols are now explained. The protocol of importance is total ordering. Causal and arbitrary ordering is achieved by relaxing the delivery constraints of total ordering and will be described in relation to total ordering. To simplify explanations, the initial descriptions assume a single group (no overlapping groups). This assumption is later removed and overlapping groups of the same protocol are described. Finally, how entities that participate in different groups simultaneously, using say an asymmetric protocol in one group and a symmetric protocol in another group, is described. Dynamic groups and fault-tolerant issues are considered later (see 3.3).

3.2.1 Symmetric Total Order in a Single Group

Consider only a single group, $g_X = \{P_1, P_2, \ldots, P_n\}$ and assume that no P_i, $1 \leq i \leq n$, ever fails or leaves g_X. This means that the initial membership view of P_i is g_X and that P_i installs no other view. So, if $V_{X,i}$ denotes the (current) membership view of P_i at any given time, then $V_{X,i} = V^0_{X,i} = g_X$. Each P_i maintains a logical clock (a counter) denoted as LC_i, that is used for numbering messages as in [Lamport78]:

- **CA1** *(Counter Advance during $send_i(m)$)*: Before sending m, P_i increments LC_i by one, and assigns the incremented value to the message number field $m.c$; and,
- **CA2** *(Counter Advance during $receive_i(m)$)*: When P_i receives m, it sets $LC_i = max\{LC_i, m.c\}$.

Based on CA1 and CA2, the following two properties can be stated:

- **pr1**: $send_i(m) \rightarrow send_i(m') \Rightarrow m.c < m'.c$; and,
- **pr2**: *for any* m, $P_j \in m.g$: $delivery_j(m) \rightarrow send_j(m'') \Rightarrow m.c < m''.c$.

Together these two properties imply that for any distinct m, m': $send(m) \rightarrow send(m') \Rightarrow m.c < m'.c$ [Lamport78].

Each P_i maintains a vector called the *Receive Vector*, denoted as $RV_{x,i}$. This vector has one integer field for every $P_j \in V_{x,i}$; this field records the counter value of the latest message received from P_j. Let $D_{x,i}$ denote the minimum value in $RV_{x,i}$: $D_{x,i} = min\{RV_{x,i}[j] \mid P_j \in V_{x,i}\}$. As $V_{x,i}$ includes P_i, $D_{x,i} \leq LC_i$ at any given time. As the underlying network is FIFO (see 3.1) messages from a given member are sent with increasing numbers and received in FIFO. Therefore, $D_{x,i} \leq LC_j$ for all $P_j \in V_{x,i}$ and P_i is guaranteed not to receive any new m such that $m.c \leq D_{x,i}$. So P_i can 'safely' deliver all received m, $m.c \leq D_{x,i}$.

- **safe1:** a received m, $m.g = g_x$, is deliverable if $m.c \leq D_{x,i}$;
- **safe2:** deliverable messages are delivered in the non-decreasing order of their numbers; a fixed pre-determined delivery order is imposed on deliverable messages of equal number.

The two *safety conditions* ensure that the received messages are delivered in total order provided they become deliverable. A received message can be guaranteed to become deliverable, only if members in $V_{x,i}$ remain *lively* by sending messages so that $D_{x,i}$ increases with time. Each member is provided with a simple mechanism, called the *timesilence* mechanism, that enables a member to remain lively by sending *null messages* during those periods it is not generating computational messages. It is assumed that this mechanism for a given P_i prompts P_i to send a null message, if no (null or non-null) message was sent by P_i in the past interval of a fixed length, say, w. Null messages contain only protocol related information (such as number, destination group identifiers etc.). When a null message

is sent or received by P_i, LC_i is advanced as per CA1 and CA2; however, when it is due for delivery, it is not supplied for processing. In brief:

- The timesilence mechanism exists solely to ensure pending (received) messages may become deliverable.

3.2.2 Symmetric Total Order in Multiple Groups

The single group assumption will now be removed, permitting P_i to be a member of more than one group. Let G_i be the set of groups P_i belongs to: $G_i = \{g_X \mid P_i \in g_X\}$, $|G_i| > 1$. Each member in the system maintains only one LC, irrespective of the number of groups it belongs to; further, this LC is advanced as per CA1 and CA2 irrespective of the group in which the member sends or receives (null or non-null) messages. Therefore the properties *pr1* and *pr2* will be true for all messages in the system. Every P_i maintains a distinct receive vector $RV_{X,i}$ for each group g_X in G_i, representing $m.c$ of the last m received from every $P_j \in V_{X,i}$. Let D_i be the minimum of all $D_{X,i}$ computed for every g_X in G_i: $D_i = min\{D_{X,i} \mid \forall g_X \in G_i\}$. Then, it is only necessary to modify the delivery condition safe1 to:

- **safe1':** a received m is deliverable if $m.c \leq D_i$.

The timesilence mechanism of P_i will operate independently for each g_X in G_i, prompting P_i to send a null message in a given group g_X, if no (null or non-null) message was sent by P_i in that group g_X for the past w time units. This ensures that $D_{X,i}$ of different g_X in G_i advance independent of each other and that the value of D_i increases with time, ensuring that any received m will eventually become deliverable.

Conditions **safe1'** and **safe2** ensure that a received m becomes deliverable for P_i only after a m', $m'.c \geq m.c$, is received from every $P_j \in V_{X,i}$ and for all g_X in G_i. These conditions, together with the timesilence mechanism can therefore cope with arbitrarily complex group structures.

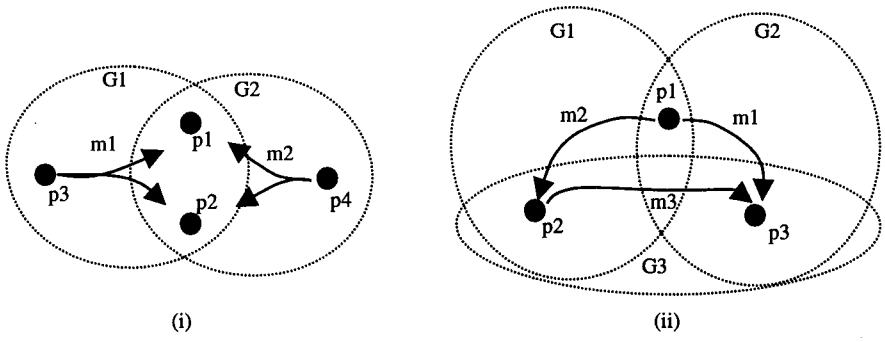

Figure 3.2 - Total ordering via symmetric protocol.

The diagrams in fig. 3.2 aid in the understanding of symmetric total ordering for overlapping groups. Consider a simple situation (fig. 3.2.i), where *m1* and *m2* are messages with the same number (say x); there is a requirement that *p1* and *p2* deliver these messages in an identical order, and before the delivery of any messages with numbers > x. Assume *p3* has a fast communication path to *p1* and *p2*, *m2* is still in transit. Conditions **safe1'** and **safe2** will nevertheless ensure that messages with a number > x are delivered only after x messages have been delivered. Fig 3.2.ii illustrates a cyclic group structure. Assume that $m1 \rightarrow m2 \rightarrow m3$, and *m1* is still in transit as *m3* is received at *p3*. Causal delivery at *p3* ($m1 \rightarrow m3$) is however guaranteed (*p3* will maintain a receive vector each for *g2* and *g3*).

3.2.3 Asymmetric Total Order Version in a Single Group

The asymmetric protocol uses one of the members of a group as a sequencer for ordering messages. Consider the case of P_i belonging to group g_x. To multicast a message m in g_x, P_i unicasts m to a member of the group, called the *sequencer*, which P_i selects out of the members in its (current) membership view of g_x using a deterministic algorithm (so members that have the same view are guaranteed to choose the same sequencer). A simple deterministic algorithm could be; each group-view held by the members of a group lists members in the same order (based on a weighting derived from their ASCII representation), the member at the head of this list is always selected to be the sequencer. The sequencer multicasts the unicast messages it receives to all members in its view in the received order and P_i delivers messages (including its own) in the order they are received from the sequencer. The sequencer, when wishing to multicast to the group, simply multicasts.

Logical clocks are still maintained for each member according to the rules CA1 and CA2; sending and receiving of unicasts update the logical clock exactly in the same manner as multicasts do. This ensures that the messages that were consecutively unicast by a given process will be

multicast by the sequencer with increasing message numbers. So, when P_i receives a multicast message m, it will no longer receive a message with numbers smaller than $m.c$ and hence P_i can deliver m straightaway. In effect, messages are deliverable as soon as they are received.

3.2.4 Asymmetric Total Order Version in Multiple Groups

Before extending the above scheme to the case where P_i can belong to multiple groups, we will observe that when P_i is not the sequencer, it disseminates its message m to the group members (not by a direct multicast as in symmetric version, but) indirectly through another member. When the sequencer multicasts m, it assigns a new $m.c$ which will be different from, and larger than, the number P_i assigned to m in its unicast. P_i cannot know the new $m.c$ of its own m until it receives m from the sequencer. Therefore P_i observes the following blocking rule when it is a member of multiple groups:

- **Asymmetric Send Blocking Rule:** A multi-group member P_i must delay unicasting of a message m (to the sequencer), until it has received (from the relevant sequencers) all the previous m', $m'.g \neq m.g$, which it has unicast.

The above rule ensures that the number given to m by P_i (and therefore by the sequencer of $m.g$) will be larger than the number given to m' by the sequencer of $m'.g$. That is, consecutive messages disseminated by P_i in different groups are guaranteed to be multicast by respective sequencers with increasing numbers.

Let $G_i = \{g_x \mid P_i \in g_x\}$ and $|G_i| \geq 1$. P_i does not maintain a receive vector as it can compute $D_{x,i}$ simply as the number of the last received message from the sequencer of g_x. It computes D_i exactly as in the symmetric version, ie., $D_i = min\{D_{x,i} \mid \forall g_x \in G_i\}$, and uses conditions **safe1'** and **safe2** for delivery.

Only the objects that are members of more than one group, need to operate the timesilence mechanism. This will ensure that the value of D_i increases with time and the protocol is lively.

3.2.5 Generic total order version

The generic protocol enables an entity P_i to execute the symmetric protocol version in one group (say g_y) and the asymmetric protocol version in another (say g_z). Let $G_i = \{g_x \mid P_i \in g_x\}$ and $|G_i| \geq 1$. Such mixed-mode working is made possible because both the protocols use the same message numbering scheme. The asymmetric blocking rule needs to be modified as follows:

- **Mixed-mode Blocking Rule:** A multi-group member process P_i must delay unicasting or multicasting of a message m, until it has received (from the relevant sequencers) all the previous m', $m'.g \neq m.g$, which it has unicast.

In addition, P_i will operate the timesilence mechanism and compute $D_{x,i}$ for each $g_x \in G_i$, depending on whether symmetric or asymmetric version is being run in g_x. It computes D_i as $D_i = min\{D_{x,i} \mid \forall g_x \in G_i\}$ and uses conditions **safe1'** and **safe2** for delivery.

3.2.6 Causal Ordering Protocol

To enable the protocol to be only causal ordering a slight amendment is required to safe2:

safe2: messages are delivered in the non-decreasing order of their numbers; *there is no need to fix a pre-determined delivery order for deliverable messages of equal number, their ordering requirements are of no importance.*

Reducing delivery guarantees from total ordering to causal ordering does not significantly increase the rate at which messages may be delivered. The simple approach of using receive vectors adopted by NewTOP ensures that messages that become deliverable are causally ordered and messages with the same numbers need to be delivered in some pre-determined order to satisfy total ordering. Therefore, it is only this process of imposing an ordering on deliverable messages of equal number that may be discarded if causal ordering alone is required.

3.2.7 Arbitrary Ordering Protocol

To enable the protocol to ensure arbitrary ordering amendments are required to **safe1**. The message delivery conditions of **safe1'** and **safe2** are not required.

- **safe1:** a received m, $m.g = g_X$ is deliverable

The safe1 described above states that messages are deliverable when they are received. When arbitrary ordering is imposed view management is supported but computational messages may be delivered in different group views by members of a group; virtual synchrony does not hold.

3.3 Introducing Dynamic Groups

How to enable groups to be dynamic and fault-tolerant is now described: ordering and liveness is preserved even if membership changes occur due to (suspected) entity failures, voluntary entity departures and new group formations. This requires every entity to operate the timesilence mechanism independently in every group in which the object is a member. This is necessary even if simple atomic delivery of messages is sufficient, since failures cannot be detected otherwise. As stated earlier, each P_i has a *group-view* process, denoted as $GV_{x,i}$, for each g_x, $g_x \in G_i$. $GV_{x,i}$ is responsible for maintaining P_i's view of the group membership of g_x. Informally, this extension has the following aspects:

1. $GV_{x,i}$ uses timeouts to *suspect* a failure of some member (P_j) that does not seem to be responding;
2. in which case $GV_{x,i}$ can initiate a *membership agreement* on P_j, the outcome of which is that either non-suspected members agree to eliminate P_j from the group view, with an agreement on the last message sent by P_j, or P_j continues to be a member and P_i is able to retrieve any missing messages of P_j.

3.3.1 Message Stability

It is necessary to ensure that a member can always retrieve a missing message from another functioning member. This in turn requires a mechanism that enables a member to safely discard a received message. To develop such a mechanism, *message stability* is defined:

- *Message Stability*: A message m becomes stable in P_i if P_i knows that all members in the current view of $m.g$ have received m.

Message stability information is piggybacked on the transmitted messages. That is, when a message m, $m.g = g_x$, is transmitted by P_i, a field $m.ldn$ (ldn: largest deliverable message number) will have the current value of $D_{x,i}$. To identify stable messages, P_i maintains a vector called $SV_{x,i}$ (Stability Vector) for each g_x. At member P_i, $SV_{x,i}[j]$ represents the latest $m.ldn$ value received from P_j. If $min(SV_{x,i})$ represents the minimum value in $SV_{x,i}$, then all m, $m.c \leq min(SV_{x,i})$ will be stable. A process can safely discard stable messages after delivery.

3.3.2 Managing Group Membership

Group-view process $GV_{x,i}$ of P_i works as if P_i is not a member of any other group. So, we can ignore the fact that P_i can be a member of more than one group, and will describe the $GV_{x,i}$ of P_i for a given g_x, dropping for convenience suffix x when no confusion is likely.

GV_i uses a communication primitive called *mcast(m)* to transmit its message m to all GV processes of $P_j \in V_{x,i}$ and the messages are delivered to (functioning and connected) destination GV processes in the sent order. GV_i has a *failure suspector* module, S_i, which monitors the liveliness of every $P_j, j \neq i$ and $P_j \in V_{x,i}$. If S_i observes that no multicast message has been received from P_j for a period $W > w$ (w = the timesilence timeout duration) then it suspects the failure of P_j and notifies GV_i of its suspicion. Application developers may assign w a value that best suits the network environment. As a guide, w may be set to larger delay than the expected transit time of a message in a network yet small enough to minimize the possibility of unfounded suspicions.

The algorithm for GV_i is given below, dropping the suffix i for all the set variables used exclusively by GV_i; these set variables are initialized to empty and a Boolean variable *consensus* is initialized to *false*, when the group g_x is formed. The algorithm describes the steps taken by GV_i, once a certain condition holds. The algorithm for GV_i has two components, *membership agreement* (for reaching agreement on entities suspected to have failed) and *view installation.*

Membership Agreement:

i. *notification* $\{P_k, ln\}$ *received from* S_i: *suspicions* := *suspicions* $\cup \{P_k, ln\}$; *mcast(i, suspect,* $\{P_k, ln\}$*);*

ii. *(j, suspect,* $\{P_k, ln\}$*) received:* ***if*** $P_k \neq P_i$ ***then*** *record the suspicion* $\{P_k, ln\}$ *of* GV_j *in gossip;* ***if*** $P_k = P_i$ ***then*** *discard the received message;*

iii. *suspicion* $\{P_k, ln\}$ *of* GV_j *is recorded in gossip* $\wedge$ *(m, m.c > ln, is received from* P_k*): mcast(i, refute,* $\{P_k, ln\}$*); /* P_i has received a message from P_k numbered > ln, so refute GV_j's suspicion of P_k; all received m of P_k, m.c > ln, can be piggybacked on the refute message */*

iv. *(j, refute,* $\{P_k, ln\}$*) received* $\wedge$ $\{P_k, ln\} \in$ *suspicions: suspicions* := *suspicions* - $\{P_k, ln\}$*; recover the missing m, m.c > ln of* P_k*; mcast(i, refute,* $\{P_k, ln\}$*);*

v. *for every $\{P_k, ln\} \in$ suspicions, suspect messages received from every GV_j of $P_j \in V - \{\{P_k \mid \{P_k, ln\} \in suspicions\} \cup failed\}$: detection := suspicions; suspicions := {}; mcast(i, confirmed, detection); consensus := true;*

vi. *(j, confirmed, $detection_j$) received $\wedge$ $detection_j \subseteq$ suspicions: detection := $detection_j$; suspicions := suspicions - $detection_j$; mcast(i, confirmed, detection); consensus := true;*

vii. *(j, confirmed, $detection_j$) received $\wedge$ $(P_i, ln\} \in detection_j$ for some ln: force S to suspect P_j; /* P_j has succeeded in suspecting P_i, so reciprocate by suspecting P_j */*

It is worth noting at this point that messages directly responsible for resolving group membership (suspicion, suspect, gossip) are multicast by members of a group and are not the responsibility of an ordering protocol.

A notification from S_i to GV_i will be of the form $\{P_k, ln\}$ - indicating that P_k is suspected to have failed and *ln* is the number of the last message P_i has received from P_k. GV_i maintains a set $suspicions_i$ where notifications from S_i are entered. GV_i also multicasts a *suspect* message *(i, suspect,* $\{P_k, ln\}$*)* to *GV* processes of all members (including GV_k) that are in its current membership view V_i. If GV_i receives confirmation that all other unsuspected members in V_i also suspect each $\{P_k, ln\}$ in its $suspicions_i$, it decides to treat each P_k of $suspicions_i$ as having failed and P_k is added to a set called $failed_i$. P_i discards any messages received from P_k and GV_k, if either $P_k \in failed_i$ or $P_k \notin V_i$. Also, once suspicion $\{P_k, ln\}$ has been added to $suspicions_i$, GV_i will keep the messages received from P_k and GV_k as pending. If suspicion $\{P_k, ln\}$ is subsequently refuted, the pending messages will be assumed to have been just received, and will be handled appropriately; if, however, suspicion $\{P_k, ln\}$ is confirmed as a failure, then the pending messages of P_k and GV_k are discarded.

Suppose that GV_j receives the message *(i, suspect,* $\{P_k, ln\}$*)* from GV_i. If $\{P_k, ln\}$ is already in $suspicions_j$, GV_j regards GV_i as yet another process that holds the same suspicion as itself; if however $\{P_k, ln\}$ is not in $suspicions_j$, it records this suspicion from P_i in $gossip_j$, but suspends judgement on it pending confirmation from its own S_j. If in the mean time P_j receives a message *m* from P_k with $m.c > ln$, then GV_j removes $\{P_k, ln\}$ from $gossip_j$ and multicasts a *refute* message *(j, refute,* $\{P_k, ln\}$*)*. When GV_i receives this refute message, it stops suspecting P_k for *ln*, and removes $\{P_k, ln\}$ from $suspicions_i$; it also initiates an attempt to recover the missing messages of P_k (a missing *m* can be piggybacked in the refute message; by definition any missing *m* is unstable, so would not

have been discarded by P_j; P_j can therefore always piggyback *m*.). After recovery of the missing message, P_i multicasts *(i, refute, {P_k, ln})* message. If GV_i ever receives a message *(k, suspect, {P_i, ln})*, it takes no action in the hope that some GV_j will refute that suspicion. When GV_i confirms all of its suspicions (condition (v)) or a subset of them (condition (vi)) into agreed failure detection, it sets the Boolean *consensus* to true. Functioning members that hold identical views and do not suspect each other, will confirm identical *detection* sets in an identical order. (A proof of this can be seen in [Mishra91].) Every agreement on a new *detection* set leads to the installation of a new view that excludes the members of the *detection* set.

View Installation:

viii. *(consensus = true): failed := {P_k | $P_k \in$ {P_k, ln}$\in$ detection}; ln_{mn} := min{ln | {P_k, ln}$\in$ detection}; **for** every $P_k \in$ failed **do** instruct P_i to discard any m received from P_k with m.c > ln_{mn} **od**; update_view(failed, ln_{mn}); **for** every $P_k \in$ failed **do** RV[k] := ∞; SV[k] := ∞; **od**; failed := { }; consensus:=false;*

The view installation component assumes the use of a primitive *update-view(F, N)* which, upon being invoked, will be executed asynchronously and will install a new view before any *m*, $m.c \geq N+1$, is delivered to P_i. The algorithm is as follows:

- *update_view(F: set_of_processes; N: integer):*
  *{ **wait until** P_i is delivered the last m, $m.c \leq N$; V:= V - F; }*

Absent or rejected messages from suspected members of the detection set prevents *D* from increasing beyond ln_{mn} and any received *m*, $m > ln_{mn}$, of *any* group will be blocked from delivery. Setting $RV[k] := SV[k] := \infty$ will allow *D* to increase more than ln_{mn} and message delivery to resume if the value of *D* has been stuck at ln_{mn}. Before setting *RV[k]* and *SV[k]* to infinity, a message *m* of a failed P_k with $m.c > ln_{mn}$ is discarded, even though it has been agreed that *m* was sent before P_k failed. This is a safety measure, necessary to preserve MD5.

After treating all the members in a given set $detection_i$ as having failed "together" and ignoring their messages with $m.c > \ln_{mn}$, GV_i calls the primitive *update_view* ($failed_i$, ln_{mn}) to install the new view, V- $failed_i$, just before any *m*, $m.c \geq ln_{mn}+1$, is to be delivered to P_i. *RV* and *SV*

are also updated to reflect the new view. As new view is installed only after the last m, $m.c \leq ln_{mn}+1$, is delivered, $m.s$ will be in the current view for any m delivered in group g_X (MD1 is met).

The diagram in fig. 3.3 is used to aid the understanding of the group membership service. Assume *Pk* multicasts *m1* in group *G1* and then multicasts *m2* in *G2*. After receiving *m2*, *pl* multicasts *m3* in group *G3*. This provides a causal relationship between *m1*, *m2* and *m3*; $m1 \rightarrow m2 \rightarrow m3$. Let *pi* and *pj* get permanently partitioned from *pk* while *m1* was been multicast, and let them not receive *m1* at all. Since $m1 \rightarrow m4$, $m4.c > m1.c$ and *pi* cannot deliver *m4* until $D_{G1,i}$ increases beyond *m1.c* which will not happen until *pk* is detected to have failed. The prolonged silence of *pk* will cause GV_i to suspect *{Pk, lnk}* for some $lnk < m1.c$, and then to reach agreement with GV_j on that suspicion. *Pk* will be removed from $V_{G1,i}$ before any m, $m.c \geq lnk + 1$, is delivered. Thus, when $m4$, $m1 \rightarrow m4$ is being delivered to *pi*, *m1.s* is guaranteed not to be in $V_{G1,i}$, if *m1* cannot be delivered to *Pi* at all.

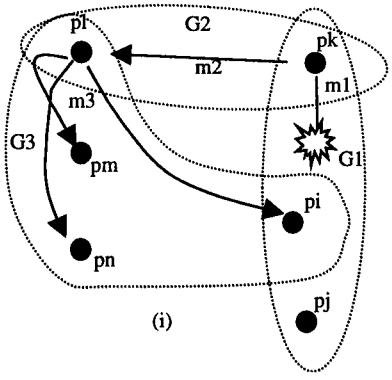

Figure 3.3 - Failure detection.

3.4 Group Formation

The group formation protocol is now described. There is an assumption that the formation of a new group can be initiated by any of the prospective group members. Selection of a potential to initiate group creation and the names of other entities that should belong to the group are dictated by higher level applications; we therefore assume that P_i (initiator) has the names of the intended members of a new group g_n. P_i must not be a member of any g_X such that $V_{x,i} = g_n$. The protocol given below has the following characteristics. A two phase protocol is used (with P_i as the coordinator) to form the group (steps 1-3). If this succeeds, then a member uses timesilence and group view process to monitor liveliness of other members (step 4); the first message P_k sends in the new group g_n is a special message *start-group* that is multicast for reaching agreement (in step 5) on the minimum value for

message number (*m.c*) with which application-related computational messages are to be multicast in g_n.

- *Step1*: P_i sends 'form group g_n' message to each intended member of g_n, inviting them to form a group; the message contains the entity-ids of the intended members of g_n.
- *Step2*: When a P_j, $j \neq i$, receives an invitation to form g_n, it diffuses this message to each intended member of g_n, piggybacking its 'yes' or 'no' decision.
- *Step3*: A 'no' message acts as a 'veto'; P_i sends its 'yes' message if it receives a 'yes ' from the rest within some time duration, else it sends a 'no'.
- *Step4*: If a $P_k \in g_n$ receives a 'yes' message from every proposed member of g_n, it activates the timesilence mechanism and a process $GV_{n,k}$ for the newly-formed g_n; the initial view $V^0_{n,k}$ is set to g_n and $RV_{n,k}$ is initialized to 0. The first message P_k sends in the new group is a special message *start-group* which contains an integer field called the *start-number* that is set to the *m.c* of the message. This number indicates P_k's proposed minimum value for message number with which application-related computational messages are to be multicast in g_n.
- *Step5*: P_k waits for the following condition to be satisfied before it can send any application related, computational message in g_n: receive a *start-group* message from every P_j in its current view $V_{n,k}$. (Note that the current view need not be $V^0_{n,k}$ due to view updates by $GV_{n,k}$ which is executing in parallel; also, P_k is not blocked from sending null messages in g_n when prompted by the timesilence). While P_k is waiting for the condition to become true, $D_{n,k}$ is not allowed to be modified except when P_k receives a *start-group* message with *start-number* larger than $D_{n,k}$, in which case $D_{n,k}$ is increased to the proposed *start-number* of the incoming message. Once all the required *start-group* messages are received, P_k sets $D_{n,k}$ to *start-number-max* = the maximum of *start-numbers* proposed by all P_j in view $V_{n,k}$; LC_k is set to *start-number-max* if *start-number-max* is larger; P_k then starts sending and delivering application-related computational messages of g_n.

To see the correctness of the group formation protocol, suppose that P_k is already a member of one or more groups when it is attempting to form a new g_n. While it is waiting for the condition of step 5 to become true, the value of $D_{n,k}$ is incremented cautiously so that P_k is not delivered any *m*, *m.c* > *start-number-max*, until that condition becomes true. Any computational message that was multicast

in g_n will have *m.c* > *start-number-max*. This ensures that P_k can be delivered the messages multicast in g_n together with those multicast in other groups, in a non-decreasing order of message numbers.

3.5 Protocol Optimizations and Extensions for Overlapping Groups

In overlapping groups a degree of message blocking is inevitable to ensure total ordering across groups (see 3.2.2). Such message blocking may be reduced for groups that are overlapped in a specific manner and the ordering protocols involved have advanced knowledge of the overlapping group structure. There are instances when an application developer will know exactly the overlapping strategies that are necessary for satisfying application requirements. Therefore, it is desirable to make available to an application developer such optimizations when they are appropriate. Following are descriptions of enhancements that may be applied to the previously described ordering protocols that seek to reduce message blocking in overlapping groups.

3.5.1 Shared Sequencer

If an entity (P_i) participates in only one group, and an asymmetric protocol is used within this group to guarantee the totally ordered delivery of messages, there is no need for P_i to block message delivery as all messages arrive from a single source (sequencer). If *Pi* assumes simultaneous membership of more than one group, irrespective of the ordering protocols used within these groups, message blocking would be incurred due to conditions **safe1'** and **safe2**. If, however, only one sequencer was present, shared by all groups that *Pi* is a member (assuming all these groups are asymmetric), then blocking may be unnecessary as all messages are arriving from a single source (shared sequencer) and are therefore totally ordered. This is because the blocking of messages is only necessary when there is the possibility of messages arriving with a lower *LC* of an already received message. This is not possible as *Pi* will only receive messages from a single source (the shared sequencer), which will always multicast a message with a higher *LC* than all preceding messages.

Overlapping groups that may benefit from a common sequencer are classified as *shared sequencer* groups. Groups classified as shared sequencer do not block the delivery of messages. However, to ensure the correctness of a shared sequencer group the two shared safety conditions must be observed:

- **shared-safety1**: Non -shared sequencer groups may not overlap with shared sequencer groups.
- **shared-safety2**: The election of a new sequencer must share the membership of the same groups as its predecessor.

Ensuring the shared-safety conditions is left to application developers. A violation of either/or both shared-safety conditions will result in the totally ordered protocol defaulting to arbitrary ordering.

3.5.2 Event Driven and Lively Groups

In the protocols described in this chapter a member is required to stay *lively* within a group to avoid being suspected of failure by other members and to ensure pending messages become deliverable. The protocols described here enable a member to remain lively by sending *null messages* during those periods it is not generating computational messages. The mechanism responsible for this action is termed the timesilence mechanism. The rate of computational messages generated within a group is application dependent. If the rate is high then the timesilence mechanism will not be generating "*null*" messages, alternatively, a low rate may result in large numbers of "*null*" messages. In situations where there is a high number of "*null*" messages compared to computational messages, the primary purpose of the timesilence mechanism reduces to failure detection.

There is a possibility of using the failure detection mechanism only when necessary to save networking resources. For example, an application may provide a service to a number of occasional clients. A conferencing style application may be structured as a group of participating clients, each client represented as a member of a group. Clients are provided with mechanisms that allow the distribution of text messages throughout the membership of the group. Clients may be geographically separated by large distances and message passing is infrequent. The benefits of detecting failure during quiet periods (no client messages) may not be any greater to the end user as detecting failure when attempting to send a text message to the group. Therefore, an application developer may classify a group as follows:

- *Lively* – timesilence mechanism and failure detection is active throughout the lifetime of a group; the duration of the timesilence period is specified at the creation time.
- *Event* – The timesilence mechanism is only active when computational messages exist within the group. Once all these messages are delivered to group members the failure suspicion and timesilence mechanisms are shutdown. The appearance of further application dependent messages reactivates these mechanisms.

3.6 Summary

The protocols and mechanisms presented in this chapter enable a group communication service that may satisfy a wide variety of application requirements. Protocols capable of two ordering types have been specified (total and causal) together with two methods for accomplishing these orderings (asymmetric and symmetric). Mechanisms have been described that are capable of handling dynamic

groups (failure detection, membership agreement, view installation). Mechanisms that ease the development and optimize the performance of a distributed application are presented (shared sequencer, event driven and lively groups). In the following paragraphs the group communication protocols of NewTOP are compared to protocols developed for use in similar works.

The Trans and Transis family of protocols [Melliar-Smith91, Amir92, Dolev93] use symmetric protocols for providing total order delivery, but are not as general purpose as NewTOP, as they rely on network level broadcast communication; further, the issue of an entity belonging to multiple groups has not been addressed. ISIS was the first system to include support for multiple groups; however the vector clock based protocols of ISIS [Birman91] become quite difficult and expensive to implement for arbitrary group structures. All previously published symmetric total order protocols require multicast messages to contain explicit information about causally preceding messages, and represent the received messages in a directed acyclic graph. The task of maintaining such a graph is much more complicated - especially for multiple groups - than the simple approach of using receive vectors adopted in NewTOP. NewTOP is able to offer this advantage because it does not attempt to precisely represent the *absence* of causal relations among multicasts as this is not essential for total order message delivery. The net effect is that NewTOP has low and bounded message space overhead (the protocol related information contained in a multicast message is small) and is relatively easy to implement even when process groups overlap in an arbitrary manner. Further, NewTOP has the capability, not available on any existing protocols, of supporting both symmetric and asymmetric protocols.

The membership algorithm of NewTOP coordinates view updates with message delivery. NewTOP maintains view consistency in the presence of (real or virtual) partitions by permitting a group of entities to partition themselves into two or more sub-groups of connected processes with the property that: (i) the functioning entities within any given subgroup will have identical views about the membership; and (ii) the views of entities belonging to different subgroups are guaranteed to stabilize into non-intersecting ones. This makes NewTOP more powerful than many other protocols [Ricciardi91, Mishra91] that can guarantee continued group operation only when the group partitions in such a way that exactly one subgroup can be uniquely identified as the primary. Due to the ability to enable partitionable operation the membership service of NewTOP is essentially similar in functionality to those of Transis [Amir92], the protocols of [Melliar-smith91, Schiper93] and Relacs [Babaoglu94].

NewTOP supports dynamic formation of new groups. The formation protocol exploits the fact that processes are permitted to belong to several groups. The group formation facility is more powerful than 'joining an existing group' facility of current protocols, as the effect of joining a group can be obtained by processes forming a new group and exiting the previous ones.

Chapter 4

The NewTOP Service

The NewTOP service is a CORBA service that implements the protocols described in the previous chapter. This chapter describes the design and implementation of this service.

4.1 Overview

The NewTOP service is a CORBA service. Clients of the service may issue a request to a group of objects with a single invocation and retrieve responses from such a request. Members of a group are CORBA objects. Clients of the service may also manage the creation, deletion and membership of object groups. Due to the varied group communication requirements that applications may place on the NewTOP service, a developer may configure the functionality of the underlying protocols supported by the NewTOP service on a per group basis. Objects may participate in more than one group simultaneously, allowing the membership of groups to overlap.

Only CORBA compliant mechanisms have been used in the implementation of the NewTOP service. This enables the NewTOP service to run on any CORBA compliant ORB, allowing applications to benefit from the interoperability and portability associated with the CORBA environment. As with any CORBA service, the NewTOP service is presented via an IDL interface. This interface specifies the methods available for application developers to allow the integration of group communication related services into their applications.

4.1.1 Enabling Client/Service Interaction

An application developer creates potential group members as CORBA objects. Such objects are presented via an IDL interface. A single group member is addressable via a single object reference (IOR). This enables the NewTOP service to identify group members and associate them to groups. A group member may also be referred to as a client of the NewTOP service.

The NewTOP service is a distributed service and achieves distribution with the aid of the *NewTOP Service Object* (NSO). Each client (group member) is allocated an NSO. Group related communications required by a client are handled by its NSO. A client, irrespective of how many groups the client participates in, requires only one NSO. The underlying ORB handles communication between a client and its NSO. Therefore, the NSO may reside within the same

address space, in a different address space, or on a different node in the network to its associated client. The most efficient configuration would be the client and its NSO residing within the same address space. Fig. 4.1 shows the communication relationships between NewTOP service clients and their NSOs.

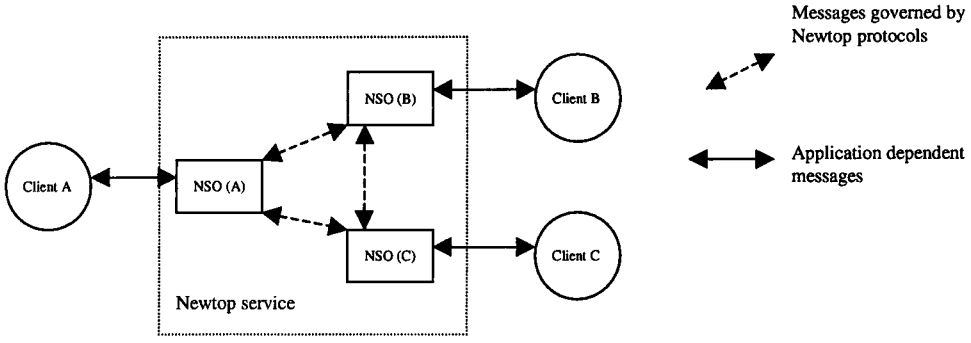

Figure 4.1 - Clients of the NewTOP service and associated NSOs.

4.2 Services

The NewTOP service consists of three services implemented by corresponding objects within the NSOs: (i) membership, (ii) invocation/multicast, (iii) group management (see fig. 4.2).

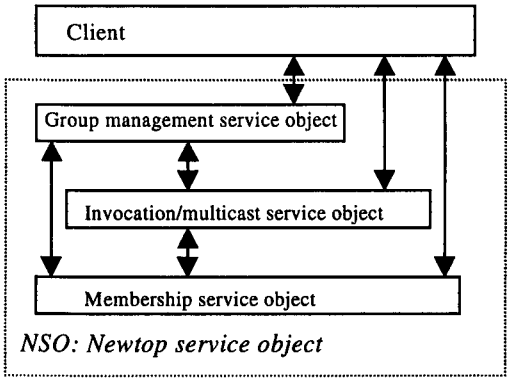

Figure 4.2 - NewTOP services.

The management service provides clients with create, delete and leave group operations. The invocation/multicast service provides four group invocation operations (wait for responses from all, from majority, from one and an asynchronous, no wait invocation). The membership service maintains the membership information and ensures that this information is mutually consistent at each member. This is achieved with the help of a failure suspector that initiates membership agreement as soon as a member is suspected to have failed. The client can obtain the current membership information by invoking the 'groupDetails' operation. Fig. 4.3 summarizes the main operations

provided by an NSO.

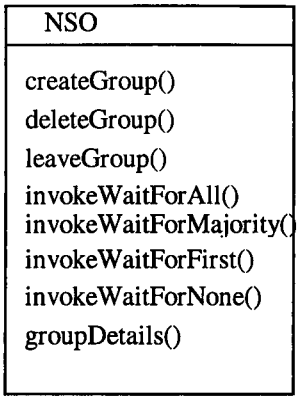

Figure 4.3 - Summary of NSO operations.

A more detailed description of each service is now presented.

4.2.1 Management Service

The management service manages the creation of new groups, the deletion of existing groups, and the change of membership for existing groups.

- *Creating a group* – The creation of a group is initiated by a client of the NewTOP service; it is assumed that the relevant NSOs have already been created. The client is required to give the group an identifier that can aid the client and the NewTOP service in differentiating between groups. This identifier should be unique and consists of a string of ASCII characters. The client is also required to supply an initial member list containing the IORs of the NSOs. The objects identified by the list are considered group members at the start of a group's life.
- *Ordering* - A client may specify what ordering guarantees are required for enabling message delivery (total, causal, arbitrary) and what style of ordering protocol is to be used (asymmetric or symmetric) within a group.
- *Type* - A group may be designated as lively or event driven. In an event driven group the timesilence mechanism (used for detecting member failure and to advance message delivery) is only active in the presence of computational messages derived from a client. In a lively group the timesilence mechanism is active all of the time.
- *Deleting a group* – A client may specify, at any time, that a group is to be deleted. The deletion of a group does not result in the deletion of the individual group members, but only of the abstract group entity. When a group has been marked for deletion all group members are told by the management service to voluntarily leave the group. The group membership service may, due to

members leaving and/or members failing, indicate to the management service that the membership of a group has reduced to a singleton. This results in the management service deleting this group.

- *Leaving a group* - At any time during the lifetime of a group a member may request to leave the group.

The underlying protocols do not support an explicit join facility. Since members are permitted to belong to several groups a similar effect can be obtained by members forming a new group and exiting the previous group. Joining a group in this manner results in the identifier of the group changing after each join is accomplished. As the joining of a group may result in computations that are application dependent (e.g., state transfer in replica groups), it is left to the application developer to implement a join facility that tackles any inconvenience associated with changing group identifiers.

4.2.2 Invocation/Multicast Service

The Invocation/multicast service manages all aspects of messages related to the delivery of client requests to object groups and server replies to clients. It is worth noting that the next chapter provides a detailed description of the protocols that the invocation/multicast service implement.

- *Invocation* - A client may issue a request in an asynchronous or synchronous style.
- *Server replies* - A client issuing a synchronous request may dictate the number of server replies to wait for (all, majority or one) and how these replies are handled by an NSO in the event of failure within the server group.
- *Protocols* - The invocation/multicast service provides the protocols for guaranteeing the ordering (total, causal or arbitrary) and delivery atomicity (with respect to group view updates) of messages.

The NewTOP service relies on the message passing capabilities of the ORB for enabling multicast communication between group members. Since, at present, ORBs only provide one to one communication, multicasting has been implemented by making invocations in turn to all the members. CORBA supports synchronous and asynchronous RPC. The asynchronous style RPC is termed *oneway* and allows an RPC to be sent while enabling the calling client to continue execution (no blocking of client). However, the CORBA specification indicates that a oneway RPC does not need to be attempted by an ORB. Some ORBs do implement oneway, some do not. For this reason, synchronous RPC has been chosen for use in the NewTOP Service. Multiple threads of execution are used to obtain parallelism and prevent client blocking.

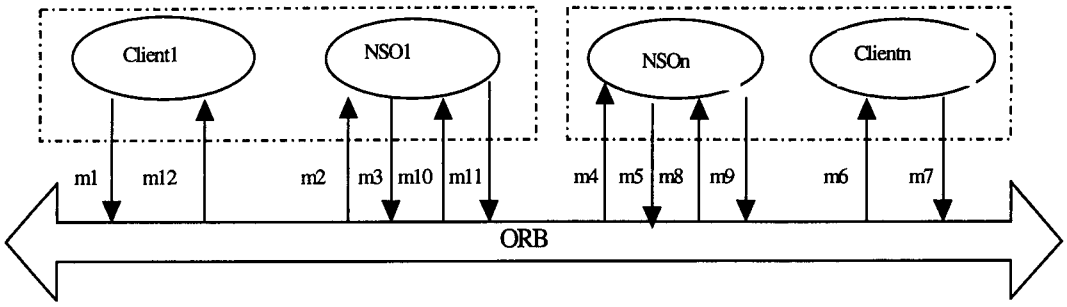

Figure 4.4 - Message interactions in a group multicast.

The principal message exchanges involved in making a group invocation are now explained. Assume a group of *n* identical objects and *object1* wants to make a synchronous group invocation on some operation of the objects; this invocation will have to be made via the group service. Fig. 4.4 shows two of these objects and their respective NSOs. The client of the NewTOP service making the invocation is required to marshall the invocation request, consisting of the name of the function and associated parameter list, into a single structure and send it to its NSO. Message 1 (*m1* for short) is such a message; *m2* is its reception. As a result, *NSO1* sends NewTOP specific messages to other NSOs; in fig. 4.4, *m3* is such a message and *m4* is its reception at *NSOn*. *NSOn* responds by composing and sending the appropriate invocation message, *m5*, to its target object (*objectn*); *m6* is its reception at *objectn*. The response from *objectn* (*m7*) is received by *NSOn* (*m8*); *NSOn* then sends NewTOP specific message (*m9*), it is received at *NSO1* (*m10*), from here *m11* and *m12* indicate the final journey back to the invoker. An NSO (such as *NSOn*) that is receiving an invocation on behalf of its target object must be able to compose the type specific invocation on the fly; this is made possible by making use of the *Dynamic Invocation Interface* (DII) feature of the ORB (in the fig. 4.4, the invocation represented by the message pair *m5*, *m8* uses DII).

4.2.3 Group Membership Service

The group membership service maintains a mutually consistent view of a group membership for each member of a group.

- *Detecting member failure* – The NewTOP service may suspect member failures with the aid of a timeout based failure suspicion protocol and/or exceptions thrown by the underlying ORB when attempting an RPC. As described in the previous chapter, suspecting a member of failure results in the execution of the membership agreement protocol; the suspected member will be removed from the group or will remain in the group with all suspicions removed. Whatever the outcome of the protocol, group members will retain mutually consistent views of the group membership.
- *Single group membership* – When membership of a group falls to singleton the group is marked

for deletion, the management service is informed and all information relating to the group is removed.

Changes in group membership are reported to the invocation/multicast service to enable pending messages to be appropriately managed (i.e., delivered or disregarded). The group membership service also provides clients with a mechanism that enables clients to gain current group views of any group which the client is a member.

4.3 Implementation Issues

This section describes issues relating to the manner in which the NewTOP service was implemented. The creation of NSOs, representing application information within messages, group transparency and a suitable threading model for ensuring the correct functioning of protocols are covered.

4.3.1 The Creation of an NSO

The function of creating NSOs is handled by an NSO factory (NSOF). An NSOF is a CORBA object, the sole purpose of which is to create NSOs and manage existing NSOs that it has created. An object wishing to participate in group communications is allocated an NSO by an NSOF. Further to this action, an NSOF indicates to a newly created NSO the IOR of the object that it is to service. This enables an NSO to issue requests (derived from other group members) on its associated client object when such an object is acting as a server in a group. The allocation of an NSO to a client object may be achieved in two ways:

- *Direct allocation* - An object issues a request to an NSOF that results in the NSOF allocating an NSO to the requesting object (fig. 4.5.i).
- *Indirect allocation* - The NSO is allocated to an object which is unaware of such an allocation. A request is issued by an object (not the object to be associated with the NSO) that results in the allocation (fig. 4.5.ii).

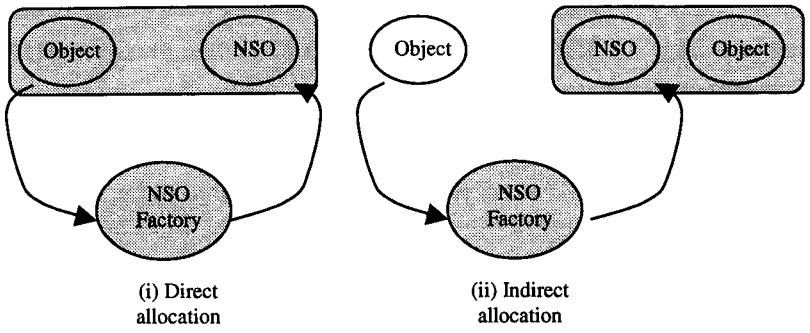

Figure 4.5 - Allocating an NSO.

Indirect allocation is used when the object that is associated with an NSO is unlikely to issue requests or retrieve group related information. An example of this is when a number of objects implement a service via a server group. These objects are not required to have knowledge of the server group, they are simply required to service client requests. Client requests manifest themselves within a server group as a request issued by an NSO via the DII. As a request issued by an NSO via the DII appears no different than a request issued by any other object it is possible to accomplish an arrangement where members of a server group are unaware of their participation in a group. In the indirect allocation approach an NSOF does not indicate to an object the IOR of their allocated NSO. The direct approach enables objects to gain the IOR of their NSO from the NSOF allowing such objects to immediately start issuing requests to object groups.

An NSOF maintains a record of all the NSOs that it has created. An NSOF provides a function that allows clients to delete selective NSOs. Further to this, a client may request a complete list of all the NSOs an NSOF has created. There may be any number of NSOFs in a system. NSOs may participate in groups together irrelevant of the NSOF that created them.

NSOs may be created as distinct processes or as threads within the process of the NSOF. By creating NSOs as distinct processes the failure of an NSOF will not hinder the functioning of the NSOs created by it. Whereas the failure of an NSOF also results in the failure of any NSOs created as threads by it.

4.3.2 Handling Application Related Message Contents by an NSO

When a client issues a request to an object group the client does so via an NSO. As a request is application dependent their structure and content are application specific. Therefore, such messages have to be formed into a structure that is suitable for handling in a generic way by NSOs. Formulating such a structure from information that may be of arbitrary type and length is commonly termed *marshaling*. The opposite of marshaling is termed *unmarshaling*.

The CORBA type *CORBA::any* together with the CORBA *sequence* data structure are used

by an NSO for carrying client requests within multicast messages. The type *CORBA::any* may hold, literally, any CORBA type, including *CORBA::any*, and any CORBA data structure. A sequence is a list of items, all items in a sequence are the same and are specified as a CORBA type. There are two types of application related messages (request and reply), each of which have their own marshaling rules that should be adhered to by application developers:

- *Request* - The function name (represented by a string) together with all parameters are transformed into values of type *CORBA::any*. They are then placed within a sequence in the order they appear in the function definition. The resultant sequence is itself transformed into a single value of type *CORBA::any*.
- *Reply* - Replies from a function call (return values and changes in parameter's values) are transformed into values of type *CORBA::any*. They are then placed in order of return (i.e., return value first followed by out parameters in the order they appear in a function definition) in a sequence. This sequence is then transformed into a single value of type *CORBA::any*.

A client is responsible for marshaling a request and unmarshalling replies to such a request. An NSO is responsible for unmarshalling a request, issuing a request at a server object via the DII and marshaling the replies. An NSO is also responsible for marshaling multiple server replies (a possible result of wait for all, wait for majority calls) into a single value. The structure of such a return value is that of a *CORBA::any* value that contains within it a sequence of *CORBA::any* values, each entry in the sequence is a server reply. The client is responsible for unmarshaling multiple server replies structured in this manner.

4.3.3 Group Transparency

Clients participating in group communications directly via an NSO are aware of server groups. The need to gain an NSO from an NSOF, marshal requests, unmarshal replies and identify the group to which their requests are to be directed results in non-transparent group communication for clients. However, servers may not be aware of any participation within a server group due to the indirect approach to NSO creation (see 4.3.1).

Providing group transparency for clients may be desirable for applications that seek high availability via the replication of a service. For example, a service may become highly available by replicating an object capable of supporting the service over a number of nodes in a network. By using a suitable replication protocol (active or passive) the service may remain available assuming that at least one object that supports the service remains correct and reachable by clients. By making the replica group transparent a client is unable to make a distinction between a service that is highly available and one which is simply provided by a singleton. This allows services to be made highly

available without the need to alter the clients of such a service.

A method for introducing group transparency to the NewTOP service is accomplished via a *proxy* object. The proxy object serves as a representation of a server group for a client and is placed between a client and its NSO (see 2.6.4). There are two ways in CORBA to implement a proxy object:

- *Hard coded per group* - An application developer codes a proxy object on a per group basis.
- *Generic* - An application developer writes a proxy object that is capable of using the DSI to assume the role of a number of different server groups.

4.3.4 Threading Model

Enabling an asynchronous message passing environment within the NewTOP service is achieved via the use of threads. Six types of threads exist within an NSO:

1. *Receive* - Accept incoming messages and place such messages into the pending buffer.
2. *Delivery* - Take messages sent by other NSOs (other group members) from the pending buffer, ensure the ordering and delivery guarantees of messages are satisfied and deliver messages to the application layer. If replies are forthcoming from the application layer, and the request that raised them is synchronous, formulate replies and multicast them to the appropriate group (via multicast thread). Alternatively, if open group message forwarding is required, create an emulated client thread and pass the message that resulted in this creation to the newly created emulated client (see next chapter for detailed description of open groups, emulated clients and message forwarding).
3. *Issue* - Take messages sent by the client/emulated client (group requests) and multicast them to the appropriate group.
4. *Emulated client* - Formulate group request and issue such a request to the appropriate group. Accept replies and formulate a reply message to send to appropriate group.
5. *Multicast* - Issues multiple RPCs to enable a multicast. If an exception is raised by the ORB indicating RPC failure during the sending of a message, the target of the RPC is suspected of failure and it's IOR is placed in the suspect buffer.
6. *Failure Detector* - Monitors the liveness of group members and the entries in the suspect buffer to determine if the group membership algorithm is to be executed.

The emulated client and the multicast threads are created when required by other threads in the NSO. There may be many emulated client threads (multiple open group requests) and many multicast threads (multiple requests for multicasts) in existence simultaneously. However, a thread of this type terminates after the completion of its task. The other threads are in singular existence during the

lifetime of an NSO and collaboratively implement the functionality supported by an NSO.

4.4 Summary

In this chapter a service has been described that enables application developers to integrate group communications into their CORBA applications. The CORBA environment has been extended to support group communications in a standard way (i.e., the service approach). This approach is the same as that adopted by OGS [Felber98a]. This ensures that applications built using the NewTOP service may benefit from the interoperability benefits associated with CORBA; there is no reliance on any group communication sub-system [Narasimhan97] or non-standard protocols that enhance the ORB [Birman93, Maffeis95]. The NewTOP service and OGS provide asynchronous message passing via different methods; OGS uses oneway function calls to achieve asynchronous message passing whereas the NewTOP service uses threads. The use of threads to enable asynchronous message passing ensures that the NewTOP service will operate correctly on ORBs that may not implement oneway function calls. Furthermore, OGS does not support overlapping groups and does not allow an application developer to choose between different types of message ordering protocol (i.e., asymmetric, symmetric).

Chapter 5

Protocols For Clients and Servers

The provision of a service may be accomplished via a group of objects/entities. Such a group is termed a *server group*. Clients may gain service from a server group (issue requests and receive replies). The overlapping group functionality of NewTOP provides a very flexible way of implementing client/server group interaction protocols. This chapter describes such protocols.

5.1 Overview of Client/Server Group Interactions

Clients may gain service from a server group via an open group or closed group approach:

1. *Closed group* - A client is considered a member of the server group and multicasts requests to each member of the server group. When message latency is high between a client and a server group (e.g., geographically separated by large distances) client requests will take far longer to service than if the server group was a singleton. As a member of the server group, a client may be required to participate in group communication protocols as a member of a server group (e.g., group management, message ordering), possibly requiring further multicasting on behalf of the client and possibly the blocking of messages. For this reason, closed groups are more appropriate when clients and a server group exist on the same LAN or neighboring LANs.
2. *Open group* - A client is not considered a member of the server group and issues requests to just a single member of the server group. Unlike the closed group, clients do not participate in group communication protocols as a member of the server group. This makes the open group approach more suitable than the closed group approach for use in wide area networks (WANs), such as the Internet, when message latency between a client and a server group may be high.

The NewTOP service is capable of supporting both the above approaches. The protocols presented here that enable open and closed group client/server interaction benefit from the existence of NewTOP. A layered approach may be used to structure these protocols together with the application and communication network. Fig 5.1 shows this layered approach and also describes the type of message passing that occurs between the layers of a client and a member of a server group. The invocation layer contains the protocols suitable for enabling open and closed group client/server interaction.

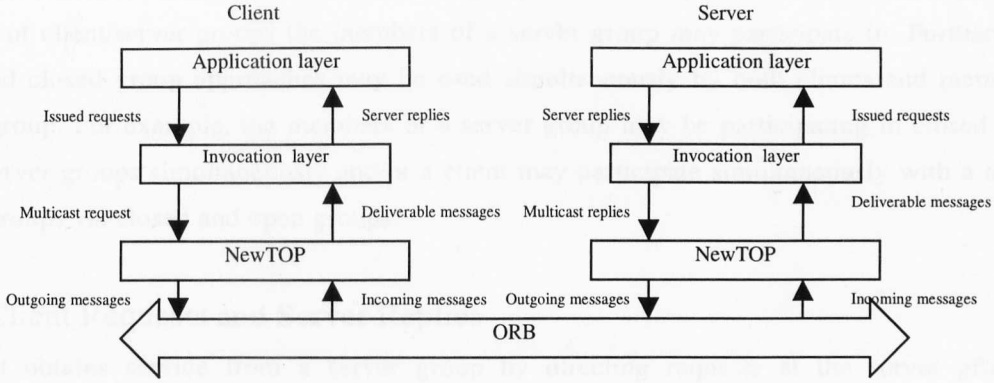

Figure 5.1 - Layering protocols.

The remainder of this section provides an overview of the techniques used by the invocation layer to accomplish open and closed client/server group interactions.

5.1.1 The Overlapping of Groups

The invocation layer achieves the open and closed group approaches to client/server group interactions via the overlapping of groups. A single group containing members that support some service is identified as a server group. Clients wishing to access services provided by a server group create a group containing themselves that overlaps with (shares membership of) the server group. A group that contains clients and servers is termed a *client/server* group. To satisfy open and closed groups, the overlapping of client/server and server groups may be achieved thus:

1. *Closed group* - Client/server group contains client and all members of the server group (fig 5.2.i).
2. *Open group* - Client/server group contains client and only one member of the server group (fig.5.2.ii).

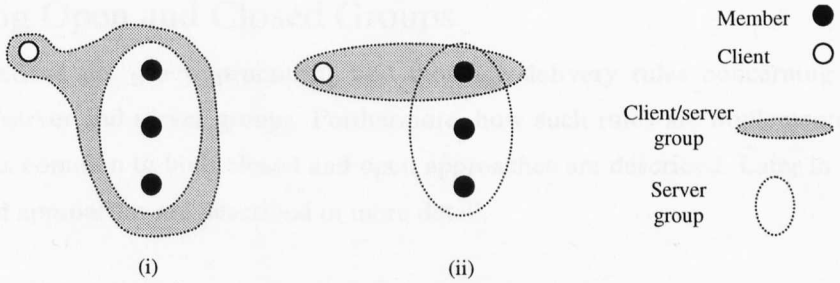

Figure 5.2 - Achieving closed and open groups.

There is no limit to the number of client/server groups a client may form. Nor is there any limit to the number of client/server groups the members of a server group may participate in. Furthermore, the open and closed group approaches may be used simultaneously by both clients and members of a server group. For example, the members of a server group may be participating in closed and open client/server groups simultaneously and/or a client may participate simultaneously with a number of server groups via closed and open groups.

5.1.2 Client Requests and Server Replies

A client obtains service from a server group by directing requests at the server group via a client/server group. Members that act as servers within a client/server group handle such requests. The underlying group communication protocols ensure ordering and delivery guarantees of client requests and server replies when servers participate in multiple client/server groups. Requests may be classified by the number of replies the issuing client is prepared to wait for:

- *One way send* - A request requires no reply. A client that issues such a request does not wait for replies and may continue processing; asynchronous message passing.
- *Wait for first* - Only wait for a reply from a single member of the server group; synchronous message passing.
- *Wait for majority* - Wait for replies from a majority of the server group; synchronous message passing.
- *Wait for all* - Wait for replies from all members of the server group; synchronous message passing.

Replies generated from client requests are sent to a client directly (closed group approach) or indirectly via a member of a server group (open group approach).

5.2 Enabling Open and Closed Groups

This section defines the group structuring and message delivery rules concerning the interaction between client/server and server groups. Furthermore, how such rules are implemented is described. Initially, aspects common to both closed and open approaches are described. Later in this chapter the open and closed approaches are described in more detail.

5.2.1 Valid Group Structures

All communication between clients and servers is accomplished via the underlying group communication service. All participants (clients and servers) are members of groups and therefore

have access to the underlying group communication protocols. Group views are maintained by the underlying group communication protocols and are available for inspection by the invocation layer.

Let G_i be the set of groups P_i belongs to: $G_i = \{g_x \mid P_i \in g_x\}$. Let $g_x = \{P_1, P_2, \ldots P_n\}$. P_i assuming a role of client in group g_x is indicated as $CP_{i(x)}$, P_i assuming a role of server in group g_x is indicated as $SP_{i(x)}$. Ensuring the validity of client/server and server group structures is accomplished via the following group structuring rules:

- **Structure rule 1** - For P_i to assume a role of client within a group g_x, P_i must be a member of group g_x.
- **Structure rule 2** - In a client/server group there exists only one client. The remainder of the membership assumes the role of server.
- **Structure rule 3** - A member of a client/server group g_x that is not a client (and therefore a server) must participate in another group g_y. The membership of g_y must contain all of the servers of group g_x. g_y is the server group.

5.2.2 Message Types and Structures

Protocols that manage client requests and server replies classify messages (delivered by the group communication layer or received from the application layer) thus:

- *Request* - a message that contains a client request (*m.req*).
- *Reply* - a message that contains a reply to a client request (*m.rep*).
- *Oneway* - A message that contains a client request but does not require a reply (*m.one*).
- *Null* - A message that contains no relevant information for the request/reply protocols (*m.null*). This type of message relates to underlying group communication protocol information (such as guaranteeing message ordering and group management issues) and is ignored by the invocation layer.

A CP_i maintains a logical clock used for numbering each request $CP_{i(LCK)}$ that is issued by the application layer. Only one *LCK* is required irrelevant of the number of client/server groups CP_i participates in. Before a request is issued *LCK* is advanced by 1. This value is added to a message containing the request and is known as the *invocation number (m.in)*. Further to this information, the name of the issuing client (*m.originator*) and the name of the client/server group ($m.g_x$) within which the message is sent attached. The two values of *m.in* and *m.originator* may be used to identify client requests and are cumulatively termed a *request identifier* (*m.id*). The uniqueness of a request

identifier is assured as member identifiers are unique and subsequent requests issued from the same client contain increasing invocation numbers. Reply messages are assigned the same request identifier as the request which resulted in their generation. The need for this uniqueness may be exemplified thus: a client *A* issues a synchronous (wait for first) request *m.req* to a server group containing two servers (*B* and *C*). *A* receives *B*'s reply to *m.req* (*m.rep*), acts upon this reply, and issues another synchronous request (wait for first) *m'.req*. *A* then receives *C*'s reply *m.rep* to the first request (*m.req*). If it was not for the unique identification that associates request and reply pairings *A* would not be able to distinguish between out of date replies and current replies (*m.rep* and *m'.rep*).

The delivery of a request, classified as synchronous, to the application layer requires the generation of a message suitable for holding replies from the application layer. If the application layer returns no reply such a message is still required to satisfy the synchronous nature of the request. In such circumstances, the part of the reply message that would contain the application reply is left blank. Alternatively, if the request is classified as asynchronous no reply message is generated. This is the case even if replies from the application layer are provided. Such replies in this scenario are discarded.

The invocation layer implements the following rules to enable a decision to be made regarding the delivery of requests and replies to the application layer:

- **Reply delivery rule** - *If* $(m.rep.originator = P_i) \wedge (m.rep.in = P_{i(LCK)})$ *then deliverable /* P_i is the originator and the LCK of P_i is the same as the invocation number of the message */*
- **Request delivery rule** - *If* $(m.req.originator \neq P_i) \vee ((m.req.originator = P_i) \wedge (SP_i \in m.req.g_x))$ *then deliverable /* P_i is not the originator or P_i is the originator and acts as a server in the group within which the message was sent */*

The above reply delivery rule may be extended to suit the four classification of request:

- **Oneway send delivery rule** - Do not deliver, even if delivery rule is satisfied.
- **Wait for first delivery rule** - Deliver when the reply delivery rule is satisfied.
- **Wait for majority delivery rule** - Deliver when the number of messages that satisfy the reply delivery rule exceeds half the number of members of the server group.
- **Wait for all delivery rule** - Deliver when the number of messages that satisfy the reply delivery rule equals the number of members of the server group.

5.2.3 Failures and Exceptions

A synchronous invocation terminates successfully if the appropriate reply delivery rule is satisfied, else the invocation may terminate exceptionally in two ways (returning a *wait_exception* message to the application layer):

1. **Group** - Membership of client/server group reduces to one (only includes issuing client) before any replies can be supplied
2. **Timing** - A period of time (specified by the application layer) elapses before a reply delivery rule is satisfied.

The group wait_exception may occur if the client has partitioned from all other functioning members of the client/server group or all other members of the client/server group have failed. In both of these cases a client will have received no replies in response to a request. A timing wait_exception occurs when the correct number of replies has not been received to satisfy a reply delivery rule within a timeout period. The application layer may specify this timeout.

The failure of a client results in the deletion of the client/server group associated to the failed client; the client/server group is no longer considered correct, structure rule 2 violated. If a client (say *A*), a member of client/server group *gx*, fails during the processing of a request issued by *A* (i.e., while *A* is still waiting for replies) then the virtual synchrony properties of the underlying group communication protocols ensure that all servers within *gx* are delivered the request or no servers in *gx* are delivered the request. In effect, the client/server group is deleted after the request from *A* is delivered or before the request from *A* can be delivered. If the request from *A* is delivered then (assuming request from *A* to be synchronous) replies may be generated by servers of *gx*. As with the initial request of *A*, the virtual synchrony properties of the underlying group communication protocols ensure that all members of *gx* receive replies to *A*'s request or no members of *gx* receive replies to *A*'s request. In effect, the client/server group is deleted after the delivery of replies or deleted before the replies can be delivered. Virtual synchrony ensures that events relating to message delivery, member failure and group deletion are viewed in the same order by every member of *gx*.

5.3 Enabling Closed Groups

In a closed group the client/server group of a client contains the client and all the service providers. This requires an extension to one of the group structuring rules:

- **Structure rule 3'** - A member of a client/server group g_x that is not a client (and therefore a server) must participate in another group g_y. The membership of g_y contains all of the servers of group g_x *and only the servers of group* g_x. g_y is the server group.

Groups that adhere to Structure rule 3' are termed closed client/server and closed server groups.

The algorithm used at the client side is thus:

```
ClosedClientSend {
        receive(m); {// from application layer}
        compute LCK; update m.originator, m.in, m.g
        multicast(m);
        if m is asynchronous → skip;
        □ m is synchronous →
                        if reply delivery rule not satisfied → skip;
                        □ reply delivery rule is satisfied →
                                receive(m); {// returned replies or exception)
                                return(m); {// return replies to application layer}
                        fi
        fi
}
```

The algorithm used for the server side:

```
ClosedRequestReceive{
        receive(m); {// from group communication layer}
        if request delivery rule not satisfied → skip;
        □ request delivery rule satisfied →
                formulate request from m;
                issue request to application layer;
                if m is asynchronous → skip;
                □ m is synchronous →
                        receive(m); {//from application layer}
                        multicast(m); {// return replies to client}
                fi
        fi
}
```

In a closed group approach the failure of group members may result in an indeterminate number of replies returned to the application layer. For example, a client issues a request to a server group that contains 7 members and associates the majority delivery rule with this request. A client may be aware that the group has 7 members when the request is issued and expects 4 replies in return. However, if 4 members fail during the processing of the request (before they have issued their replies), resulting in membership reducing to 3 members, then the majority delivery rule will return when only two replies have been received. A further complication occurs when replies have been received from members that have subsequently failed before a reply delivery rule may be satisfied. To allow the application layer to dictate the type of behavior exhibited by a reply delivery rule in the presence of member failures a delivery rule may be classified as:

- **D1** - The group view before the issue of a request is used when deciding if a delivery rule is satisfied. Replies received from members that may no longer appear in the current group view are considered valid.
- **D2** - The current group view is used when deciding if a delivery rule is satisfied. Replies received from members that no longer appear in the current group view are considered valid.

When a client issues a request and associates a D1 rule with such a request a reduction in the membership of the group view of the issuing client before the delivery rule is satisfied may result in a timing wait_exception. In the example described previously (7 servers reducing to 3 servers, members failing before replies may be sent) there is no possibility of ever receiving 4 replies, therefore, the reply will block until a timing wait_exception is raised.

5.4 Enabling Open Groups

Open groups require the redirection of messages from a receiving SP_i to members of a server group and an extension to one of the group structuring rules:

- **Structure rule 3"** - A member of a client/server group g_x that is not a client (and therefore a server) must participate in another group g_y. The membership of g_x is limited to two (one client one server). The membership of g_y contains all of the servers (one server in this case) of group g_x. g_y is the server group.

Groups that adhere to Structure rule 3" are termed open client/server and open server groups.

In open client/server groups requests are directed at only a single server. Therefore, a mechanism that will propagate such messages throughout the server group and collect replies ready for returning to a client is necessary. This mechanism is described, with reference to fig 5.3, as follows:

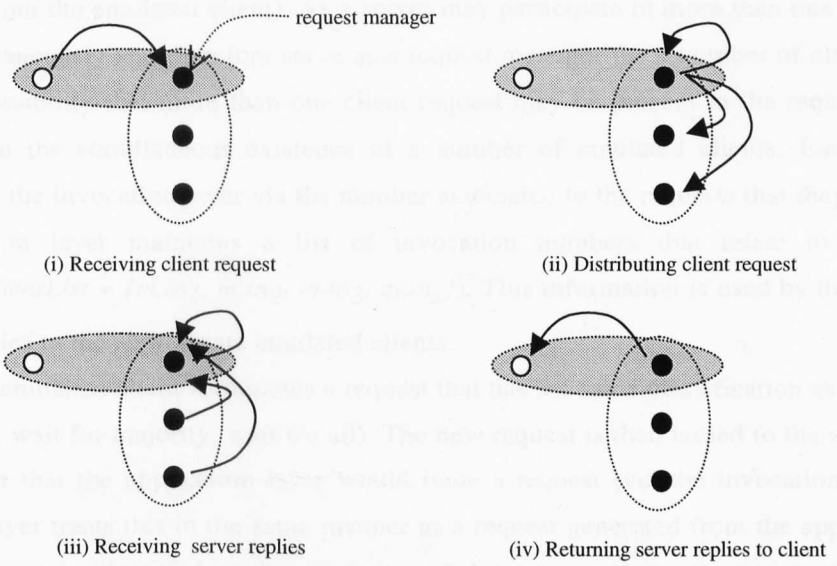

Figure 5.3 - A mechanism for handling client requests for open groups.

i. *Receiving client request* - A request (*m*) sent within a client/server group is received by a server. This server is considered to be the *request manager* for this particular client request.

ii. *Distributing client request* - The request manager ensures that *m* is reorganized into a request *m'* and issued as a request within the server group. The difference between *m'* and a request issued in the closed group fashion is that *m'* must also be delivered to the issuing client (the request manager).

iii. *Receiving server replies* – Each member of the server group multicasts replies within the group. For clarity only the messages destined for the request manager are shown.

iv. *Returning server replies to client* - Server replies are gathered and returned to the client.

This section continues with more detailed descriptions of the open group request/reply handling mechanism.

5.4.1 Redirecting Messages Containing Client Requests

The message flow involved in client request redirection at a request manager is shown in figure 5.4. When the invocation layer of a request manager is delivered a message from the group

communication layer that contains a client request sent within an open client/server group the message is not delivered to the application layer. Instead, the invocation layer creates a process thread that acts as a client for the purposes of redirecting the client request (*m1*). This process is known as an *emulated client.* The invocation layer passes the client request (*m2*) to the emulated client. The invocation layer records the value of *LCK+1* (the invocation number that will be attached to a request emanating from the emulated client). As a server may participate in more than one open client/server group simultaneously and therefore serve as a request manager for a number of client/server groups, there is a possibility that more than one client request may be present in the request manager. This will result in the simultaneous existence of a number of emulated clients. Emulated clients are identified by the invocation layer via the number associated to the requests that they issue. Therefore, the invocation layer maintains a list of invocation numbers that relate to emulated clients (*EmulatedClientList* = $\{m.in_1, m.in_2, m.in_3, m.in_x\}$). This information is used by the invocation layer to direct replies to the appropriate emulated clients.

The emulated client formulates a request that has the same classification as the original client request (e.g., wait for majority, wait for all). The new request is then issued to the server group in the same manner that the application layer would issue a request (via the invocation layer) (*m3*). The invocation layer treats this in the same manner as a request generated from the application layer; the request manager is identified as the originator of the new message, the name of the server group replaces the name of the client/server group in the $m.g_x$ field of the message and a new invocation number is associated to the message. Finally, the invocation layer uses the group communication layer to multicast the request to the server group (*m4*).

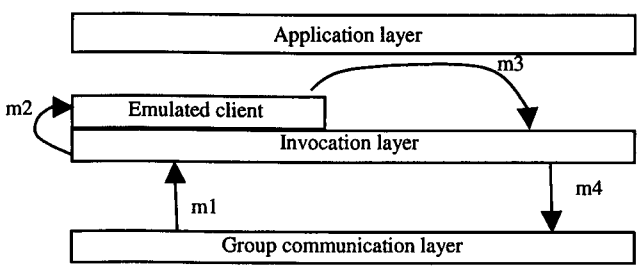

Figure 5.4 - Client request redirection at a request manager.

A client request that is classified as oneway will result in the deletion of the emulated client once *m3* has been issued.

5.4.2 Handling Requests Issued by an Emulated Client

Servers that are delivered messages from their group communication layers that contain requests generated by an emulated client are handled in the same manner as requests generated by the application layer of a client in the closed group approach. However, in the closed group approach the invocation layer does not deliver to the application layer if the member identified in the originator field of the message (the issuing client) is itself. In the open group approach this is desired. The request delivery rule identified in 5.2.2 ensures this type of operation.

5.4.3 Handling Replies Associated to Emulated Client Requests

The reply delivery rule identified in 5.2.2 is only suitable for replies destined for the application layer. This is because the reply delivery rule assumes only one possible recipient (the application layer). However, replies may be destined for an emulated client (of which there may be many) or the application layer. For this reason the reply delivery rule needs to be updated for members of open server groups:

- **Open group reply delivery rule** - *((m.rep.originator = P_i) $\wedge$ (m.rep.in = $P_{i(LCK)}$)) $\vee$ ((m.rep.originator = P_i) $\wedge$ ($P_i \in$ EmulatedClientList)) /* P_i is the originator and the LCK of P_i is the same as the invocation number of the message or P_i is the originator and there exists an emulated client that has the same identifier as the message invocation number */*

As in the case of the original reply delivery rule, the open group reply delivery rule may be extended to cover the four classifications of client requests. The classification of the open group reply delivery rule is the same as that of the reply delivery rule associated to the request issued within the client/server group that resulted in the creation of the emulated client.

5.4.4 Returning Server Replies to a Client

After the reply delivery rule has been satisfied server replies are delivered by the invocation layer to the appropriate emulated client (emulated client is specified by the invocation numbers associated with such replies). On receiving these replies the emulated client formulates a single reply message and places all the server replies received from the invocation layer within this message. The message identifier of the client/server request that resulted in the creation of the emulated client is associated to the reply message as is the name of the client/server group.

Once the group communication layer has delivered the reply to the invocation layer the invocation layer observes the original reply delivery rule when determining if the reply is deliverable. As there are only two members in an open client/server group the synchronous classification

extensions to the reply delivery rule do not hinder a single reply delivery. Hence, the D1 and D2 types of delivery rule do not apply. As the classification of the reply delivery rule specified by the issuing client is also adhered to by the request manager when propagating a request, the client may still receive a number of replies within the returned message. This facilitates the ability to gain multiple replies via the open group approach.

5.4.5 Client and Server Side Algorithms

For completeness the algorithms for client and server side operations for open group client/server communications are described. The ClientSend algorithm described previously is used by the issuing client in the client/server group.

```
OpenRequestReceive{
        receive(m); {// from group communication layer}
        if request delivery rule not satisfied → skip;
        □ request delivery rule satisfied →
                formulate request from m;
                create emulated client;
                update emulated client list; {// add new emulated client ID to list}
                deliver(m); {// deliver m to newly created emulated client}
                if m is asynchronous → skip;
                □ m is synchronous →
                        if reply delivery rule not satisfied → skip;
                        □ reply delivery rule satisfied →
                                deliver(m1); {// deliver replies to emulated client}
                        fi
                fi
        fi
}

EmulatedClient{
        receive(m1); {// from OpenRequestReceive}
        create m2 from m1; {// create request to forward to server group}
        update m1.originator, m1.in, m1.g;
        multicast(m1); {// multicast to server group}
        if m1 is asynchronous → skip;
                □ m is synchronous →
                        receive(m); {// receive replies from application layer}
```

```
                    multicast(m); {// return replies to client}
        fi
}
```

5.5 Optimizations to Open Group Structures

This section describes methods that may be employed by an application developer to reduce message blocking (arising when a protocol enforces some ordering and delivery guarantee) and/or reduce the volume of messages.

5.5.1 Restricted Open Group Structure

A client/server group used to accomplish open group communications always has two members (client and member of server group - Structure rule 3"). Therefore, a client request, originating from a client/server group, received by a server (SP_i) within such a group may be instantly deliverable if there is no possibility of a causal relationship existing between such a message and other messages received by SP_i from other groups. This will allow a client request to be received by the invocation layer without the need to block its delivery at the group communication layer. Message delivery and ordering guarantees are known to be satisfied without the need for message passing round completion in other groups that overlap directly, or indirectly, with the server group that SP_i is a member. An application developer may take advantage of this scenario by classifying an open server group as *restricted.* Fig 5.5.i identifies a suitable overlapping structure where the server group may be considered restricted. The existence of only a single request manager within a server group ensures that all messages generated outside a server group are distributed throughout the server group via a single point of entry. The request manager may immediately distribute throughout the server group, and deliver to the application layer, a message (say *m1*) received from a client/server group as there is no possibility that another member of the server group has received a message (say *m2*) from a client/server group such that $m2 \rightarrow m1$. However, when more than one member of a server group may act as a request manager (fig 5.5.ii) the immediate delivery and distribution of messages is not possible as there is the possibility that more than one request manager may simultaneously deliver different messages, ignoring any causal relationship that may exist between them. The rule governing the validity of a restricted open server group is fairly trivial:

- **Restricted group rule** - Only one member of an open server group may act as a request manager. Electing a single request manager in a restricted server group would be achieved via a deterministic algorithm. A simple deterministic algorithm could be; each group-view held by the

members of a group lists members in the same order (based on a weighting derived from their ASCII representation), the member at the top of this list assumes the role of request manager.

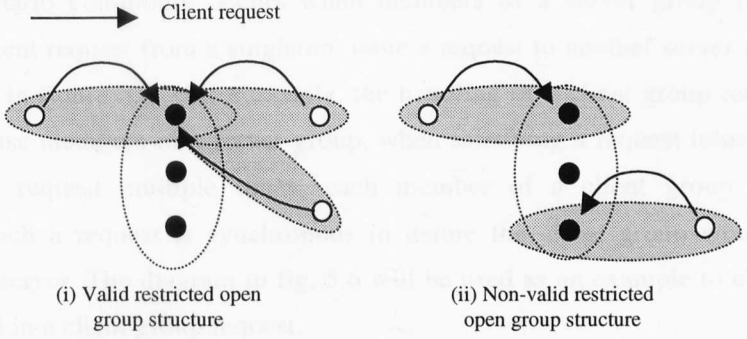

Figure 5.5 - Valid and non-valid restricted open group structures.

5.5.2 Asynchronous Message Forwarding

A client request is propagated by the request manager in the format it was received. Synchronous client requests will be propagated by a request manager in a synchronous fashion. This enables a client to gain multiple server replies via the open group approach. When a client simply requires a single reply the propagation of client requests in an asynchronous style is sufficient (assuming the request manager is capable of returning a reply to the issuing client itself). This reduces message volume per client request in the open group approach as members of a server group are not required to reply to a client request emanating from an emulated client. With regard to fig. 5.3, step (iii) would be eliminated.

The rule governing asynchronous message forwarding is thus:

- **Asynchronous group rule** - Client requests are issued as asynchronous to every member of the server group except the request manager dealing with the client request. The request manager is sent the request in a wait for one style.

The above rule ensures that an emulated client only receives one reply, which can then be sent (as a reply message) back to the original client. This approach is suited to the support of passive replication protocols (see 2.4.3).

Combining the restricted open group and asynchronous message forwarding approaches provides suitable support for passive replication protocols. The request manager may assume the role of the primary: receiving, processing and replying to client requests. The remainder of the server group are passive members, receiving (but not necessarily acting upon) client requests.

5.6 When Clients are Groups

A group that issues a request is termed a client group and such a request is termed a client group request. This scenario commonly occurs when members of a server group (say *gx*), during the processing of a client request from a singleton, issue a request to another server group (say *gy*). This scenario is shown in figure 5.6. Unfortunately, the handling of a client group request is not a trivial task. This is because members of a server group, when satisfying a request issued by a client group, receive the same request multiple times; each member of a client group issues the request. Furthermore, if such a request is synchronous in nature the client group would receive multiple replies from each server. The diagram in fig. 5.6 will be used as an example to clarify the number of messages involved in a client group request.

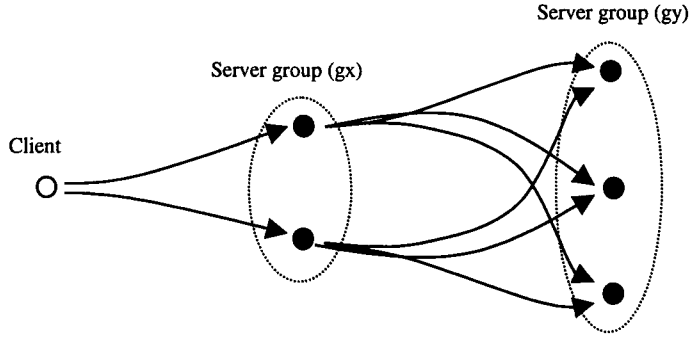

Figure 5.6 - When clients are groups.

Assume a client issues a synchronous request to a server group (*gx*, 2 members). During the processing of this client request each member of *gx* issues a synchronous request to another server group (*gy*, 3 members). At this point *gx* assumes the role of a client group. As *gx* has two members, each member of *gy* will receive 2 requests (6 requests received in total). Each member of *gy* will reply to each request issued by *gx*. As 6 requests have been received at *gy* then each member of *gx* will receive 6 replies, a total of 12 replies. To aid clarity in the diagram, the replies have not been shown. This scenario is an example of a *nested operation*; an operation that results in the invocation of yet another operation or, in the case of client groups, the invocation of one object group leading to the invocation of another object group [Narasimhan97]. The problem of reply message duplication is compounded when a chain of nested operations involving client groups is present

The remainder of this section describes in detail how client group communications are accomplished by the NewTOP service while solving the problems associated with message duplication. In the following descriptions group members are assumed to act deterministically; two

members of a group (say, *m1* and *m2*) will output identical messages in identical order, provided they are delivered input messages in identical order.

5.6.1 Client Group Requests

The initial problem to overcome is that of reducing the multiple copies of the same client group request arriving at each member of a server group. This is achieved, in part, by adhering to the following client group rule:

- **Client group rule 1** - Client groups interact with server groups via the open group approach. This requires each member of the client group forming a client/server group with a member of a server group (see fig 5.7).

The above rule ensures that each request sent by a member of a client group to a server group is sent as a single message as opposed to a multicast. This reduces multicasts to unicasts. The next step is to ensure that a client request is represented within a server group by a single message. This is achieved with the aid of two further rules:

- **Client group rule 2** - Multiple requests emanating from a client group that cumulatively represent a single client group request are all handled by the same request manager.
- **Client group rule 3** - A group exists that consists of all the members of a client group and the request manager of the server group. This group contains no other members.

Client group rule 2 ensures that a single request manager receives requests issued by members of a client group (that represent a single group request) and client group rule 3 enables a request manager to realise such requests are the result of a client group request. The diagram in fig. 5.7 aids in the following description that identifies the importance of these two rules in allowing a request manager to decide which messages to forward and which to discard. The group described in client group rule 3 is termed a *client monitor group*. A request manager (of say, *gy*) receiving requests from members of a client group (say *gx*), via open client/server groups, expects requests from all members, excluding itself, of the client monitor group (say *gz*).

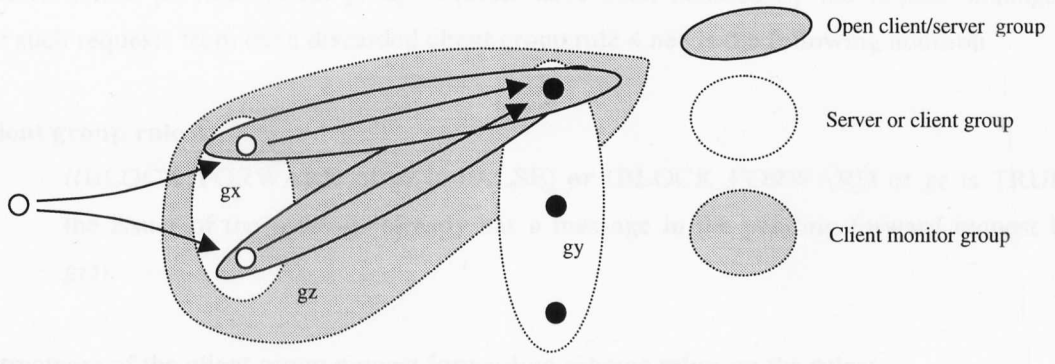

Figure 5.7 - Reducing client group requests from multicasts to unicasts.

The request manager associates to each client monitor group that it participates in a Boolean variable BLOCK_FORWARD. Initially, before any requests have been received from the client group by the request manager, BLOCK_FORWARD is set to FALSE. A request manager that deems requests suitable for forwarding stores such requests in the *pending forward request list.* Each monitor client group has associated with it a pending forward request list. The following rule aids the request manager in determining if a request from a client group is to be forwarded.

- **Client group rule 4** - Add a message to a pending forward request list **iff**
  - i. Message received from client/server group **and**
  - ii. Membership of client server group is a true subset of a monitor client group (say *gz*) **and**
  - iii. BLOCK_FORWARD of *gz* is FALSE

If client group rule 4 is satisfied (i.e., message added to pending forward request list of *gz*) then the request manager sets the BLOCK_FORWARD variable of the monitor client group which is associated to the forwarded request (*gz*) to TRUE. A message is taken from the pending forward request list and forwarded by the request manager in the order which they were placed on the list. The request manager may simultaneously handle group requests that were sent by different client groups (messages from different pending forward request lists). However, with regard to requests on the same pending froward request list, only one request may be handled at a time by a request manager (i.e., when a request is synchronous in nature replies must be gathered and returned to a client group (say, *gx*) before further replies from *gx* may be handled). The request manager forwards the request in the usual manner (i.e., via emulated client). Client group rule 4 is adequate for handling synchronous requests. In practice, client group rule 4.iii ensures that only one message at a time may reside on the forward request list. When requests are asynchronous in nature it is possible a client group request

may arrive before previous client group requests have been handled by the request manager. To prevent such requests from been discarded client group rule 4 needs the following addition:

- **Client group rule 4** -
  iii. ((BLOCK_FORWARD of *gz* is FALSE) **or** (BLOCK_FORWARD of *gz* is TRUE and the issuer of the message already has a message in the pending forward request list of *gz*)).

The correctness of the client group request forwarding scheme relies on the rules:

- **Client group rule 5** - The individual member requests emanating from a client group that cumulatively represent a client group request are totally ordered.
- **Client group rule 6** - client/server groups created by an application developer for the purposes of enabling client group requests should not be used to enable non-client group requests.

The diagram in fig. 5.8 identifies the importance of client group rule 5. Two clients each multicast a request simultaneously to group *gx*. Each client request results in *gx* issuing a synchronous request to group *gy* (say *req3* and *req4*). If requests are not totally ordered each member of *gx* may issue *req3* and *req4* in an arbitrary order (e.g., *M1* may issue *req3* before *req4* whereas *M2* may issue *req4* before *req3*). Such ordering of requests may result in requests not been forwarded. For example, if the request manager receives *req3* from *M1* before *req4* from *M2* then *req3* will be forwarded. Now, assume that *req3* has been dealt with and the request manager is ready for accepting requests (this will be described fully in the next section). The request manager now receives *req3* from *M2* before *req4* from *M1*. This will result in the forwarding of *req3* again. Furthermore, *req4* is discarded and never actually gets forwarded. By totally ordering requests, we ensure that both members of the client group always multicast requests in the same order. For example, *M1* and *M2* send *req3* followed by *req4,* or visa versa, as they were delivered *req1* and *req2* in the same order, ensuring both requests are forwarded.

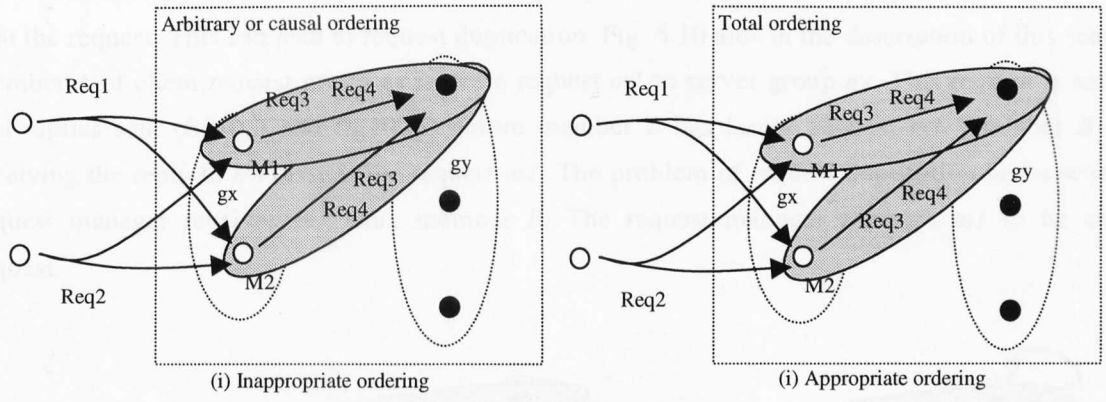

Figure 5.8 - Ensuring all requests are forwarded.

Client group rule 6 guarantees that all requests emanating from a client group are actually client group requests. Individual members of a client group (say, *gx*) are inhibited from issuing requests at a server group (say, *gy*) that are not part of a client group request when *gx* is a client group of *gy*.

5.6.2 Client Group Replies

Replies received by a request manager (fig. 5.9.i) that are the result of a client group request are sent as reply type messages back to the members of the issuing client. This is achieved via client/server groups (fig.5.10.ii). These client server/groups are the same as the client/server group initially used to send the client group request.

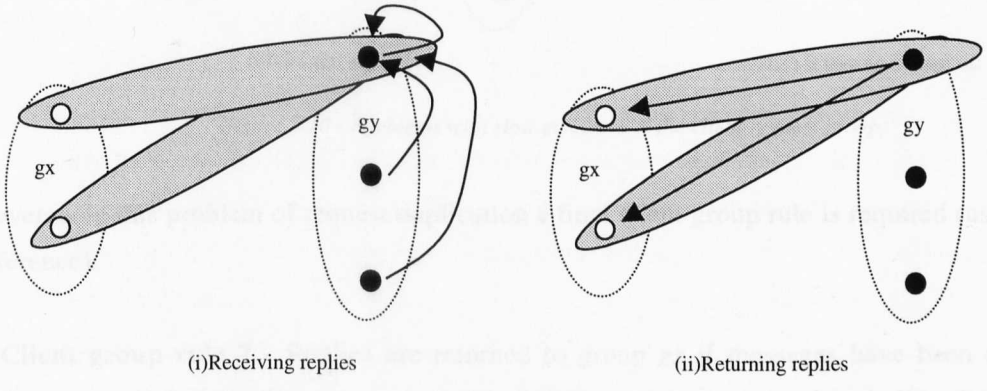

Figure 5.9 - Returning replies back to a client.

The completion of a request (sending replies back to the client group for synchronous requests or simply forwarding an asynchronous request) results in BLOCK_FORWARD been set to FALSE.

There may be instances when a client group member receives a reply before it has actually sent the request. This can lead to request duplication. Fig. 5.10 aids in the description of this scenario; member *A* of client request group *gx* issues a request *m1* to server group *gy*. This request is satisfied and replies sent (fig10.ii and fig10.iii) before member *B* has issued request *m1*. Member *B* (after receiving the reply to *m1*) issues the request *m1*. The problem of request duplication arises when the request manager receives *m1* from member *B*. The request manager assumes *m1* to be another request.

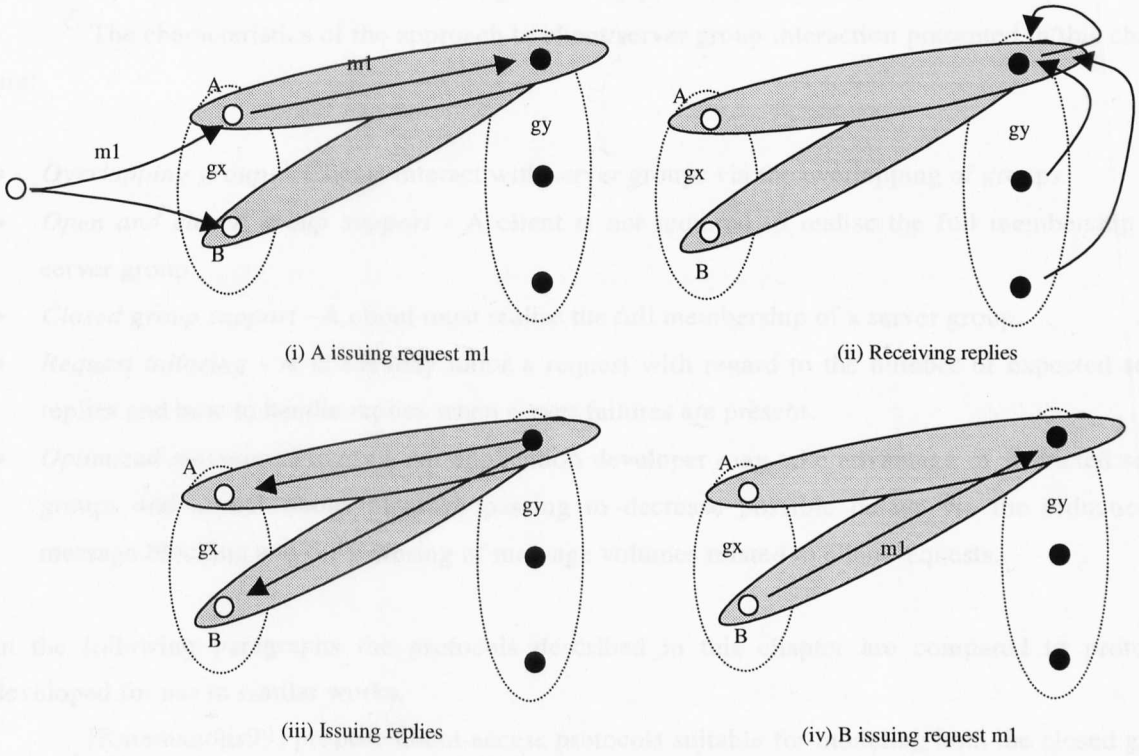

Figure 5.10 - Problems with slow members of the client request group.

To overcome this problem of request duplication a final client group rule is required (using fig. 5.7 as a reference):

- **Client group rule 7** - Replies are returned to group *gx* if messages have been received from every member of monitor group *gz* (excluding request manager and including the message that turned BLOCK_FORWARD to TRUE from FALSE).

5.7 Summary

In this chapter protocols suitable for enabling clients to issue requests to and receive replies from server groups in a number of ways have been presented. Furthermore, clients may interact with server groups in an open or closed group fashion.

The invocation protocol design exploits the properties of the NewTOP protocol suite. NewTOP provides: a number of message ordering and delivery guarantees within a virtual synchronous environment, failure detection, partitionable operation, overlapping groups, a choice between symmetric and asymmetric message ordering protocols.

The characteristics of the approach to client/server group interaction presented in this chapter are:

- *Overlapping groups* - Clients interact with server groups via the overlapping of groups.
- *Open and closed group support* - A client is not required to realise the full membership of a server group.
- *Closed group support* - A client must realise the full membership of a server group.
- *Request tailoring* - A client may tailor a request with regard to the number of expected server replies and how to handle replies when server failures are present.
- *Optimized message delivery* - An application developer may take advantage of restricted server groups and asynchronous message passing to decrease possible delays via the reduction of message blocking and the lowering of message volumes related to client requests.

In the following paragraphs the protocols described in this chapter are compared to protocols developed for use in similar works.

[Karamanolis99] propose client-access protocols suitable for modeling both the closed group and open group approaches. These protocols are specifically designed to satisfy the requirements of applications that use the state machine approach for service replication (active replication). The closed group approach is similar in functionality to the closed group approach presented in this chapter, which is initially presented in the Isis group communication service [Birman93]. However, the open approach advocated by [Karamanolis99] is significantly different from the open approach presented here.

The protocols presented in this chapter enable open group access via group communication protocols (fig. 5.11.i). In the approach presented by [Karamanolis99] a client does not use a group communication protocol when communicating with a group (fig 5.11.ii). A client issues a request via an invocation layer which directly communicates with a peer invocation layer of one of the servers of a server group. This server then distributes (multicasts) the client request throughout the server group

via a group communication service. Only the server that originally received the client request replies. Replies from other servers are filtered in their invocation layers, preventing them from been sent.

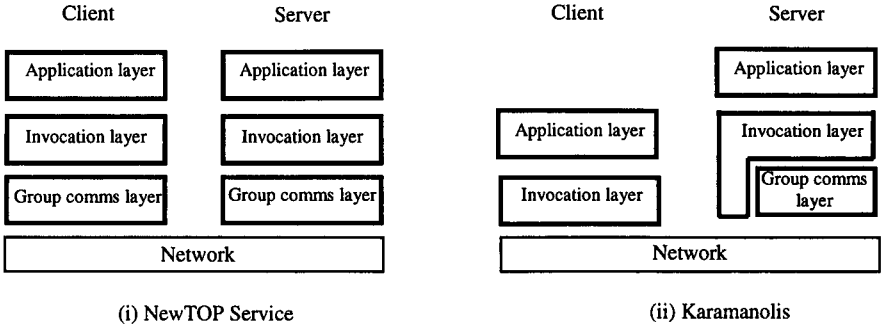

Figure 5.11 - Different approaches to open group approaches.

The functionality provided by [Karamanolis99] is similar to that provided by the open group approach with asynchronous message forwarding presented in this chapter. However, as the group communication protocols do not aid the sending of a client request to a member of the server group, there is no possibility of ensuring client requests can be causally ordered. Consider the diagram in fig. 5.12.i. A group *gx* consists of two members (*A* and *B*). *B* issues an open group request to *gy* (*m1*). *B* then multicasts a message to *gx* (*m2*). During the processing of *m2 A* issues an open group request to *gy* (*m3*). In the [Karamanolis99] approach the order with which *m1* and *m3* are handled by *gy* is always arbitrary (whichever arrives at the server first). However, the protocols presented in this chapter (if required) may ensure that *m1* is handled before *m3* by *gy*. Fig 5.12.ii describes how overlapping groups may be used to accomplish the same effect as that shown in fig. 5.12.i, yet still ensure that messages are delivered with respect to the causal relationship that exists between them (i.e., m1 → m2 → m3). Client/server groups *gz* and *gw* are used to enable open group requests by *A* and *B* respectively.

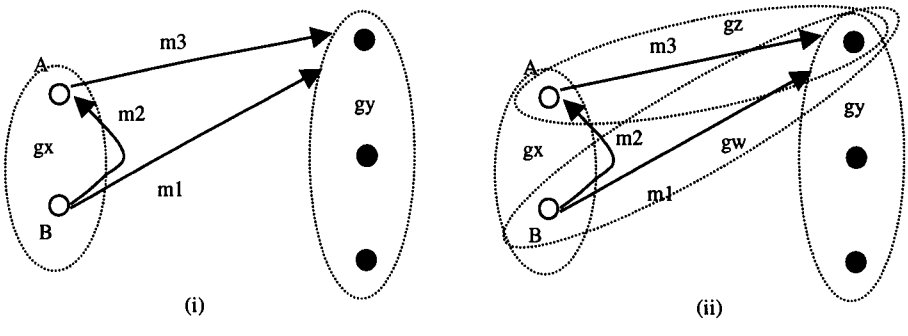

Figure 5.12 - Ordering of related client requests.

The need to ensure that the causal ordering between *m1* and *m3* is adhered to by *gy* may be exemplified thus (related to fig. 5.12):

Assume group *gx* represents a group of companies (*A* and *B*) that trade with each other. A bank provides an account which is used to enable the transfer of monetary funds between the companies *A* and *B*. Furthermore, this account is made highly available (replicated) and is presented in the diagram by group *gy*. *B* wishes to pass funds to *A*. *B* deposits funds into the account (*m1*) and informs *A* that money is waiting (*m2*). *A* attempts to retrieve funds from the account (*m3*). To ensure that *m3* may be satisfied (funds are available), *m1* must be delivered before *m3* (the causal relationship between *m1* and *m3* must be realized by *gy*).

I know of only one other group communication service that attempts to solve the "when clients are group" problem; Eternal [Narasimhan97]. Eternal places dependency on the underlying network services to enable such communications, limiting the use of the system to a LAN. The approach described here has no such dependencies. Furthermore, Eternal is unable to support open groups, client requests are enabled via closed group mechanisms.

Chapter 6

Performance of the NewTOP Service

In this chapter the performance results obtained from a series of experiments conducted with the NewTOP Service are presented. During these experiments the throughput of groups and the time taken for a client request to be satisfied were monitored. Experiments are described followed by the results of such experiments.

Experiments are performed in LAN and WAN environments. To the best of my knowledge, other Middleware group communication services have not been the subject of experiments in a WAN environment. Therefore, this chapter presents the first set of figures that indicate the performance of a Middleware group communication service in a WAN environment.

6.1 Experiments

This section describes the experiments in detail. The different types of experiments are first described, followed by the environment within which the experiments were carried out. The experiments are classified into two scenarios:

1. *Request/Reply* - A client issues a request to multiple servers and waits for their replies; this represents a commonly occurring scenario when a service is replicated, fig. 6.1.i.
2. *Peer Participation* - All the members are regularly multicasting by using the asynchronous method invocation operation; this represents a commonly occurring scenario when the purpose of an application is to share information between members, fig. 6.1.ii.

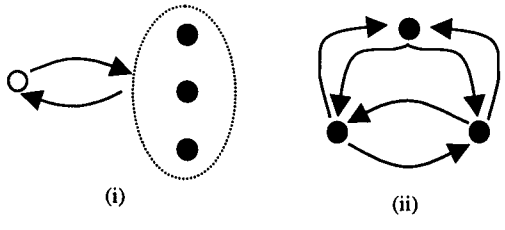

Figure 6.1 - Request/Reply and peer participation scenarios.

These two scenarios are treated separately in the experiments presented here. However, it is possible for applications to combine the two scenarios in the provision of services.

The performances of the asymmetric and symmetric total ordering protocols are of interest in all experiments and the open and closed methods for enabling clients to gain services from a server group are of interest in the request reply experiments. These experiments aim to identify the appropriate protocol configuration for request/reply and peer group scenarios (i.e., open or closed group approach to client/server group interactions and what ordering protocol is suitable: asymmetric or symmetric).

6.1.1 Request/Reply Scenario

The request/reply experiments presented here aim to identify the suitability of the NewTOP service for supporting object replication. The NewTOP service alone does not provide all the mechanisms required to satisfy object replication. For example, issues related to state transfer and the election of a primary in passive replication schemes are not addressed. However, the NewTOP service does provide the application developer with group communications suitable for a replication service:

- *Active replication (1), (fig. 6.2.i)* - The closed group approach (see 5.3). Clients multicast directly to every member of the server group. Each server processes client requests in the same order. A client may receive responses from all servers.
- *Active replication (2), (fig. 6.2.ii)* - The standard open group approach using synchronous message forwarding (see 5.4). Clients multicast indirectly to every member of the server group via any group member. Each server receives client requests in the same order. A client may receive responses from all servers.
- *Passive replication, (fig. 6.2.iii)* - The restricted open group approach using asynchronous message forwarding (see 5.5). All clients multicast indirectly to every member of the server group via the same server. Each server receives client requests in the same order. A client may receive a response from only one server, the primary (the request manager).

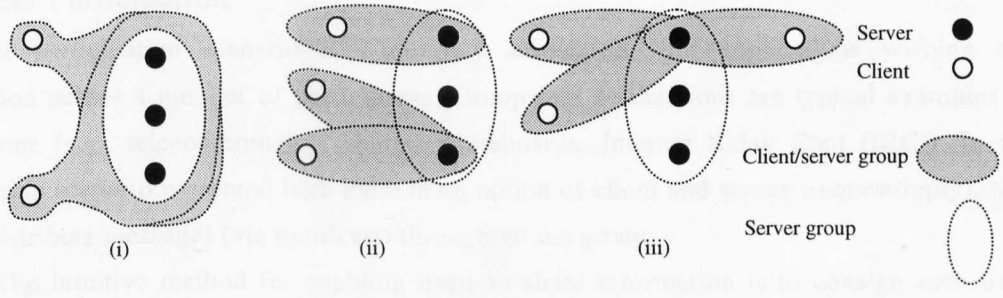

Figure 6.2 - The closed and open group approach to client/server interaction.

In these experiments measurement of the time taken to service a client request by a server group has been taken. In the asymmetric closed group experiment the shared sequencer optimization technique was used (see 3.5.1). In the asymmetric standard and restricted open group experiments only a single sequencer was used in the server group and all open client/server groups were symmetric. Due to open client/server groups only having two members the use of a sequencer would be wasteful as the redirection of messages by a sequencer within such groups would not aid the ordering of messages. The use of symmetric open client/server groups together with asymmetric open server groups displays the ability of the NewTOP service to allow members of multiple groups to use asymmetric and symmetric ordering techniques in different groups simultaneously.

During the experiments a client issues a single request and waits for the appropriate number of replies before issuing another request. In the closed group and standard open group scenarios the client waits for all server replies. The client may have waited for just a single reply or a majority of replies. However, a decision was made to wait for all replies. In the restricted open group scenario asynchronous message forwarding was used. This resulted in the client receiving just a single reply from the server group.

Clients issue requests as frequently as possible; as soon as a client has been serviced by a server group a client issues another request. Clients are run simultaneously. This ensures different client requests are produced simultaneously. Client numbers are increased 1 through 20. At each of these increments each participating client is timed for 1000 requests. This timed figure is then divided by 1000 to determine how long a single request takes (average/mean) per client. The sum of these client request times are then divided by the number of clients to gain the average time taken to service a request from any client. This figure is then used to derive the throughput of the server group on a per second basis.

At the application level a server is a pseudo random number generator. Random numbers are always three digits long. A client request is a request for a random number. The server group consisted of three members in all experiments with all groups designated as event driven (see 3.5.2).

6.1.2 Peer Participation

The peer participation scenario is commonly associated with applications wishing to share information across a number of participants. Groupware applications are typical examples of such applications (e.g., teleconferencing, shared whiteboards, Internet Relay Chat (IRC)). In the peer participation scenario presented here there is no notion of client and server (request/reply). Members simply distribute messages (via multicast) throughout the group.

The intuitive method for enabling users to share information is to consign such users to a single group.

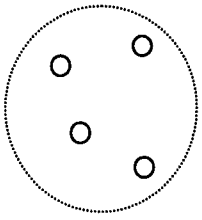

Figure 6.3 - Peer participation.

Fig. 6.3 highlights the group structure used in the peer participation experiments. Of particular interest is the performance of the asymmetric and symmetric ordering protocols. Due to the nature of the application requirements (only a single group) there is no notion of open/closed client access and so these are not of issue here.

All members issue multicasts as frequently as possible. Performance is measured by assessing how long a multicast takes to become deliverable at all members within a group from the time of the multicast's issue. The time taken for 1000 multicasts from each member to become deliverable at every other member of the group is measured. This results in a figure per client (as all clients are multicasting). Each of these figures are then divided by a 1000 to gain a figure for the time taken for a single multicast. The resultant figures are then summed to allow a throughput figure for the group to be gained.

At the application level the body of the message consists of a CORBA string type of 100 characters in length. The group was designated as lively (see 3.5.2).

6.1.3 Environment

In every case, group members reside within the same address space as their NSOs. The NewTOP service and the application objects were written in C++. The two different environments that were used in the experiments are described.

- *Local Area Network* - The system consists of 9 Pentium PCs running Red Hat Linux 2.0.34, each with 64 megabytes of RAM, connected together using 100 Mbits fast Ethernet.
- *Internet* - The network is global. Machines may be geographically separated by large distances. The machines used during the experiments were located in Newcastle (United Kingdom), London (United Kingdom) and Pisa (Italy). These machines were Pentium PCs also running Red Hat Linux 2.0.34 (as above).

The distribution of group members was thus:

- Request/reply
  - *LAN* - No two servers appeared on the same machine. Clients were equally distributed on the same LAN.
  - *Internet(1)* - Servers were located on the same LAN in Newcastle. No two servers appeared on the same machine. Clients were equally distributed between London and Pisa.
  - *Internet(2)* - Servers were geographically separated and clients were geographically separated between Newcastle, London and Pisa.
- Peer participation
  - *LAN* - Members were distributed over the LAN.
  - *Internet* - Members were distributed between Newcastle, London and Pisa.

6.2 Results

In this section the results of the experiments are presented. To enable an analysis of the figures presented here an initial experiment to derive the RPC time for two CORBA objects that do not communicate via the NewTOP Object Group Service are presented. To further enhance analysis of request/reply scenarios, figures gained from an experiment that involved a non-replicated server and a number of clients are presented.

All figures are presented in milliseconds. Throughput may be read as per second.

6.2.1 CORBA RPC

The server used in this experiment is a CORBA object that simply returns a pseudo random number of three digits when requested to do so by a client (the client is also a CORBA object of the same type as the server). The experiment consisted of one client and one server. The client issues a request and waits for a reply (synchronous communications) via CORBA RPC. The client issues requests as frequently as possible; as soon as a request is serviced another request is issued.

Without NewTOP Object group Service	*Timed request*	*Throughput*
Client and server on distinct nodes in LAN	0.9	1111.11
Client in Pisa and server in Newcastle	78.0	12.82
Client in London and server in Newcastle	81.0	12.34
Client in Pisa and server in London	86.0	11.62

Table 6.1 - Performance of CORBA RPC.

6.2.2 Request Reply, Non-Replicated Server

The same type of CORBA object as in the CORBA RPC experiment was used (pseudo random number generator). However, communications were achieved via NSOs. Clients issue requests and wait for a reply before issuing another request. Clients issue requests as frequently as possible. The results of these experiments are shown in fig. 6.4.

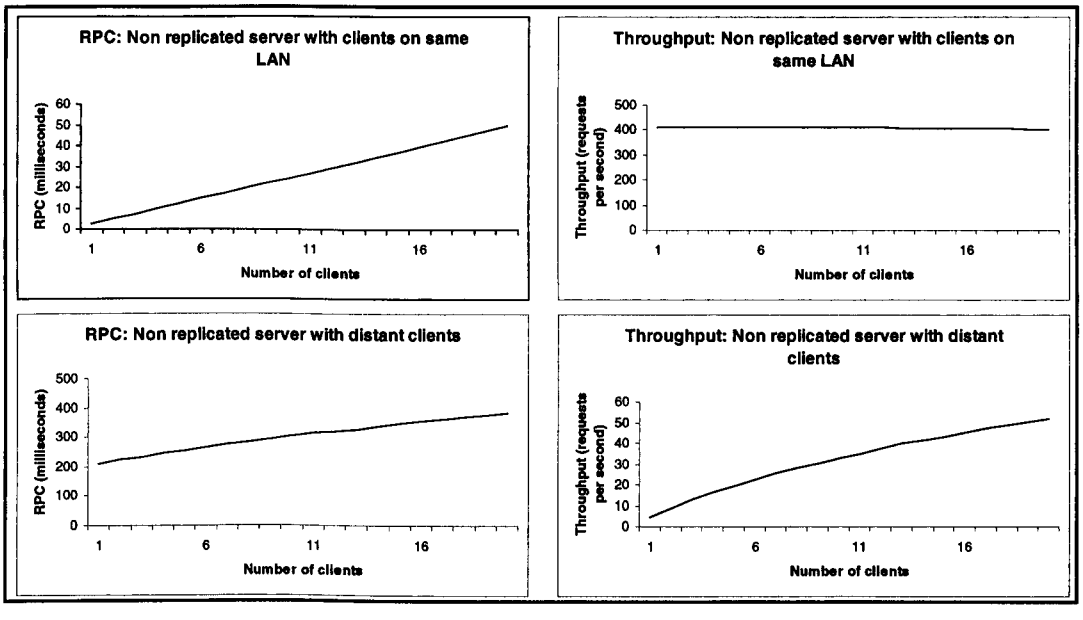

Figure 6.4 – Performance of non-replicated server.

The first observation to be made is that a single client making an RPC via the NewTOP service was approximately half the performance of a single client making an RPC without the NewTOP service in the LAN environment. This drop in performance may be explained by the manner with which an

NSO manages messages. This message passing process is shown in fig. 6.5 and is now described. A more detailed description of how client requests are handled is given in chapter 5.

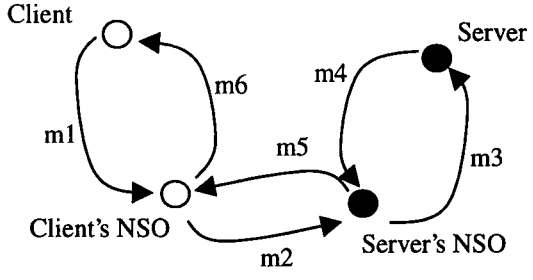

Figure 6.5 - Message passing between NSOs and application objects.

When an NSO receives a message from the application level it is queued in a multicast pending list. A thread is then created which handles the multicasting of queued messages. In the case of these experiments a request by a client for a random number is received by the client's NSO (*m1*). The client's NSO then multicasts this message to the replica group (*m2*). The server's NSO receives this message and queues this message as pending. Messages in the pending queue that satisfy ordering and delivery guarantees are then delivered to the NSO's associated application object (random number generator object) (*m3*). This delivery takes the form of an invocation. Results from the invocation are then queued by the server's NSO in the multicast pending list (*m4*). A thread is then created which handles the multicasting of messages held in this list. Finally, the client's NSO receives this return message (*m5*) and queues it in the messages pending list, awaiting delivery back to the client object of the NSO (*m6*). Assuming the client (server) NSO to be in the same address space as the client (server), request-reply message pairs m1-m6, m3-m4 will not generate any network traffic. On the other hand, message m2 as well as message m5 are each a CORBA RPC. The expected availability of asynchronous messaging and multicast services in next generation ORBs (see 7.3.2) will remove the main source of the inefficiency.

The percentage drop in performance for point-to-point communications indicates that the majority of overhead incurred by using the NewTOP service is due to the need for two RPCs as opposed to just a single RPC. There is a slight anomaly which is assumed to be processing and/or connection management overheads.

Another observation to be made from this experiment is that throughput for LAN decreases as the number of clients increases, whereas, in the Internet environment the throughput rises as the number of clients increases. There are two factors governing the throughput of a service:

1. The rate at which client requests arrive at a server.
2. The time taken to process a client request by a server.

Assuming processing time and message latency remain constant; if a server, after satisfying a client request, experiences some idle time before the next client request is received then the throughput of the server has the potential to rise. However, if a server, after satisfying a client request, experiences no idle time before the next client request is received then the throughput of the server may not rise. In practice, the throughput will decrease due to the increased processing demands made by underlying network protocols when managing client requests.

The server in both experiments (LAN and Internet) is the same. Therefore, the time taken by a server to process a client request is the same. This indicates that it is the rate at which client requests arrive at a server that results in a far lower throughput for the server servicing clients over the Internet compared to the server servicing clients over the LAN. As message transit times have been shown to be far greater over the Internet than those over a LAN (see 6.2.1), client requests arrive more infrequently at the Internet server than they do at the LAN server. This results in server idle time in the Internet server and explains why throughput is far lower than that of the LAN server.

As client requests are synchronous and a server services such requests sequentially, the only way to increase the frequency of messages arriving at a server is to increase the number of clients. In the LAN experiment increasing client numbers, even from 1 to 2, has a negative effect (all be it very small) on throughput . This indicates that throughput does not have the potential to rise and has reached it's maximum at around 410 requests per second. As the server in the Internet experiment is the same as that in the LAN experiment, the Internet server has the potential of reaching a throughput of 410 requests per second. Increasing the number of clients does substantially increase throughput. However, the extra processing required by the communication protocols to manage substantial numbers of clients may result in the throughput of the Internet server not reaching 410 requests per second irrespective of client numbers.

6.2.3 The Restricted Open Group Approach (Compared to Non-Replicated Server)

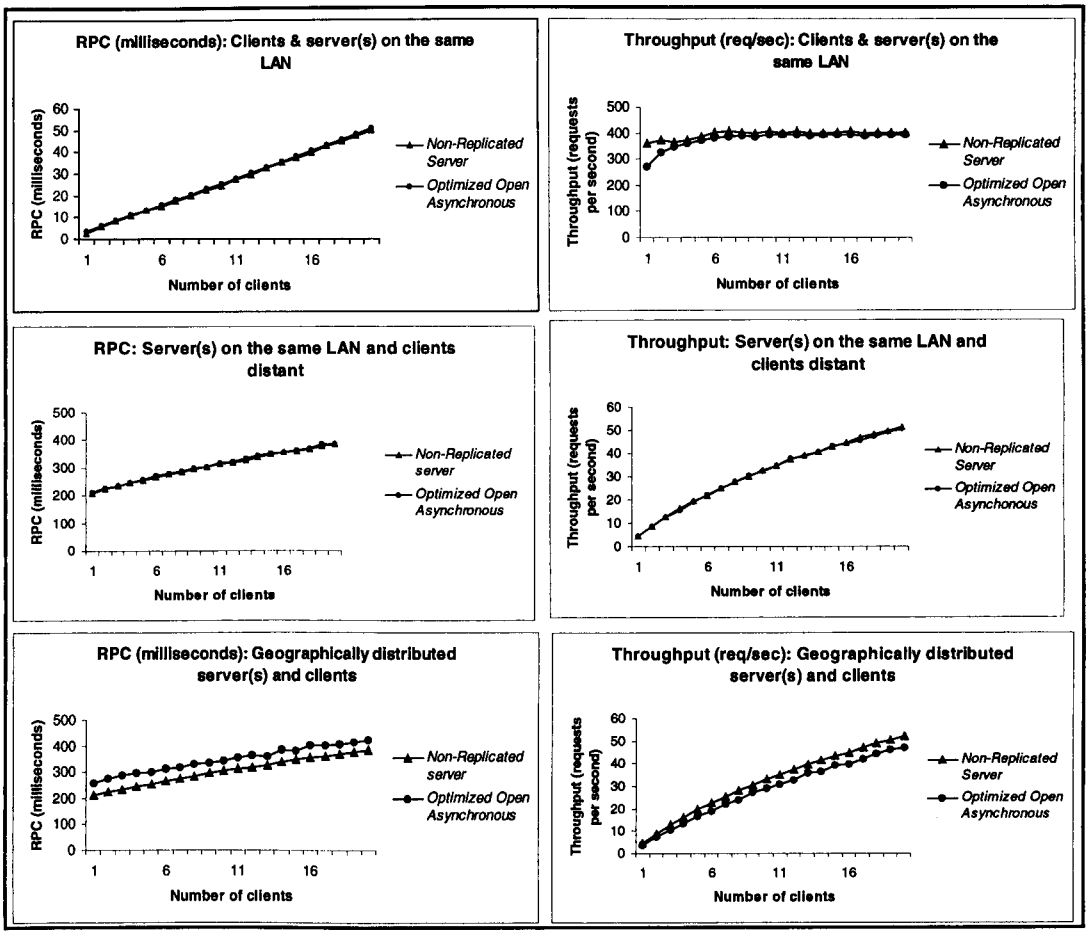

Figure 6.6 – Comparing the performance of non-replicated server and replicated server.

The next set of experiments to be considered were those relating to a group invocation requiring a reply from a single server (wait for first). These experiments present the performance of the scheme depicted in fig. 6.2.iii incorporating the restricted open group with asynchronous message forwarding (e.g., passive replication). The server group used the asymmetric ordering protocol in these experiments. However, the performance gained from using asymmetric instead of symmetric was found to be minimal (see 6.2.6). This will enable a comparison of performance between group invocation with that of non-replicated invocation discussed in the previous subsection: in both the cases, the client invokes a single server; the only additional work required for group invocation is that the request manager is required to forward the request to all the members. Since this is performed asynchronously, there is an expectation that the performance of the restricted open group invocation to closely match that of the non-replicated invocation. This is indeed the case, with the role of the

sequencer, request manager and primary all undertaken by the same group member. Fig. 6.6 presents the figures relating to these experiments.

6.2.4 The Closed Group Approach

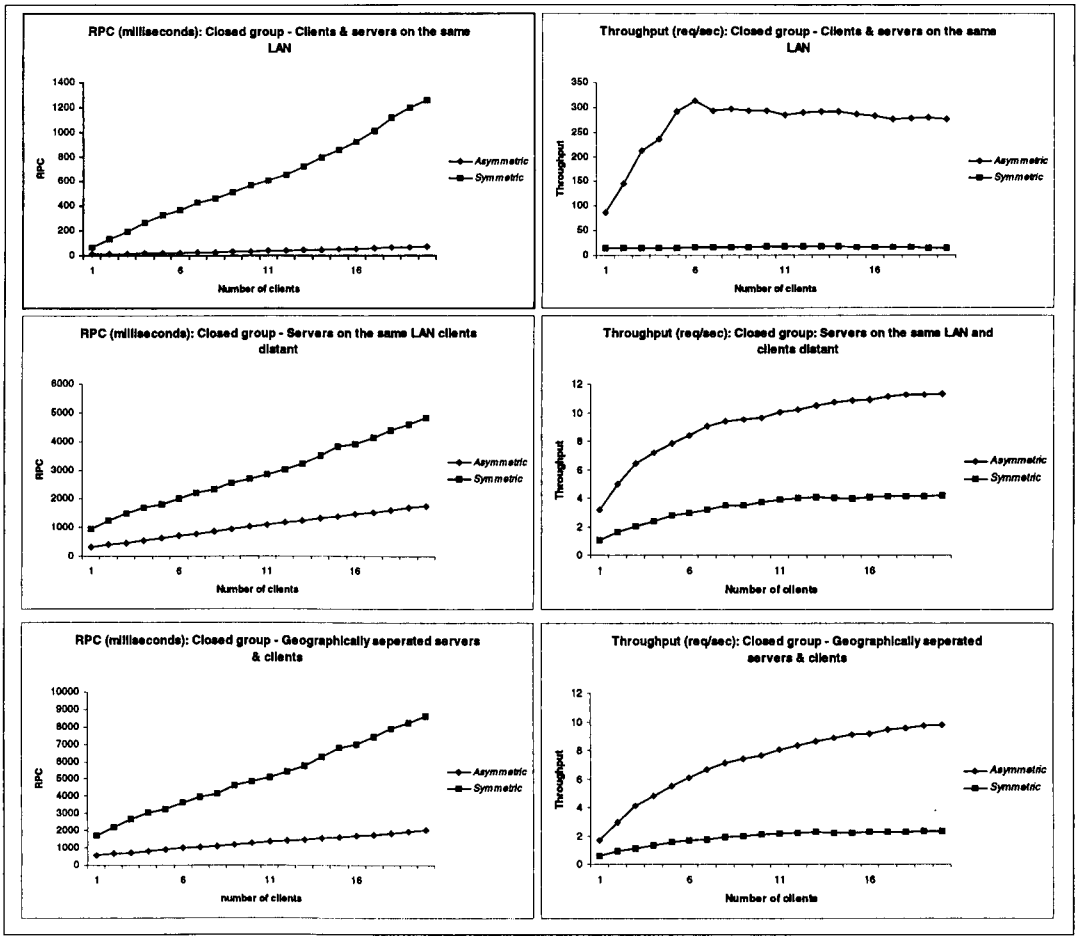

Figure 6.7 – Comparing asymmetric and symmetric protocols in the closed group approach for request/reply experiments.

The symmetric protocol blocks message delivery until each member of a group mutually agrees on the order of message delivery. Such agreement is accomplished via further message passing between group members. All message passing using a symmetric protocol is achieved via multicasting. The asymmetric protocol, aided by a shared sequencer, requires no message blocking and no additional message passing to determine message ordering. Furthermore, only the sequencer multicasts, greatly reducing the volume of messages compared to the symmetric approach.

Due to no requirement to block messages and no additional message passing required to ensure message ordering the asymmetric protocol outperforms the symmetric protocol in both the LAN and Internet experiments. However, the difference in throughput between asymmetric and symmetric is far less in the Internet experiments compared to that in the LAN experiment. This is because the time taken to service a request is dominated in both protocols by the slow Internet connection between clients and server group. As a client receives each server reply in a different multicast, so a client has to wait for three messages to be sent over the Internet in both the asymmetric and symmetric protocols. This is the reason why the performance of the non-replicated Internet server of the previous experiment is 3 to 4 that of the asymmetric protocol here. In addition to receiving three replies in distinct multicasts the symmetric approach has to receive three further multicast messages, required for message ordering, (one from each server). This identifies why the performance of the asymmetric protocol is approximately 3 times that of the symmetric protocol in the Internet experiment.

The time taken for a server group to collectively process a client request is different in the symmetric and asymmetric protocols. The blocking of messages and the extra message passing required to guarantee message ordering results in a server group using the symmetric protocol taking far longer than a server group using the asymmetric protocol to satisfy a client request. This is evident in the LAN experiment, where the symmetric throughput is far lower than that of the asymmetric throughput. Furthermore, the asymmetric throughput rises for the first 5 clients whereas symmetric throughput remains static. This slight rise may be attributed to the doubling of a client request's transit time; a client request first arrives at the sequencer and is then multicast to the group. This increase in message transit time may be sufficient to ensure that peak throughput requires 5 clients for the asymmetric protocol in the LAN experiments.

6.2.5 The Standard Open Group Approach

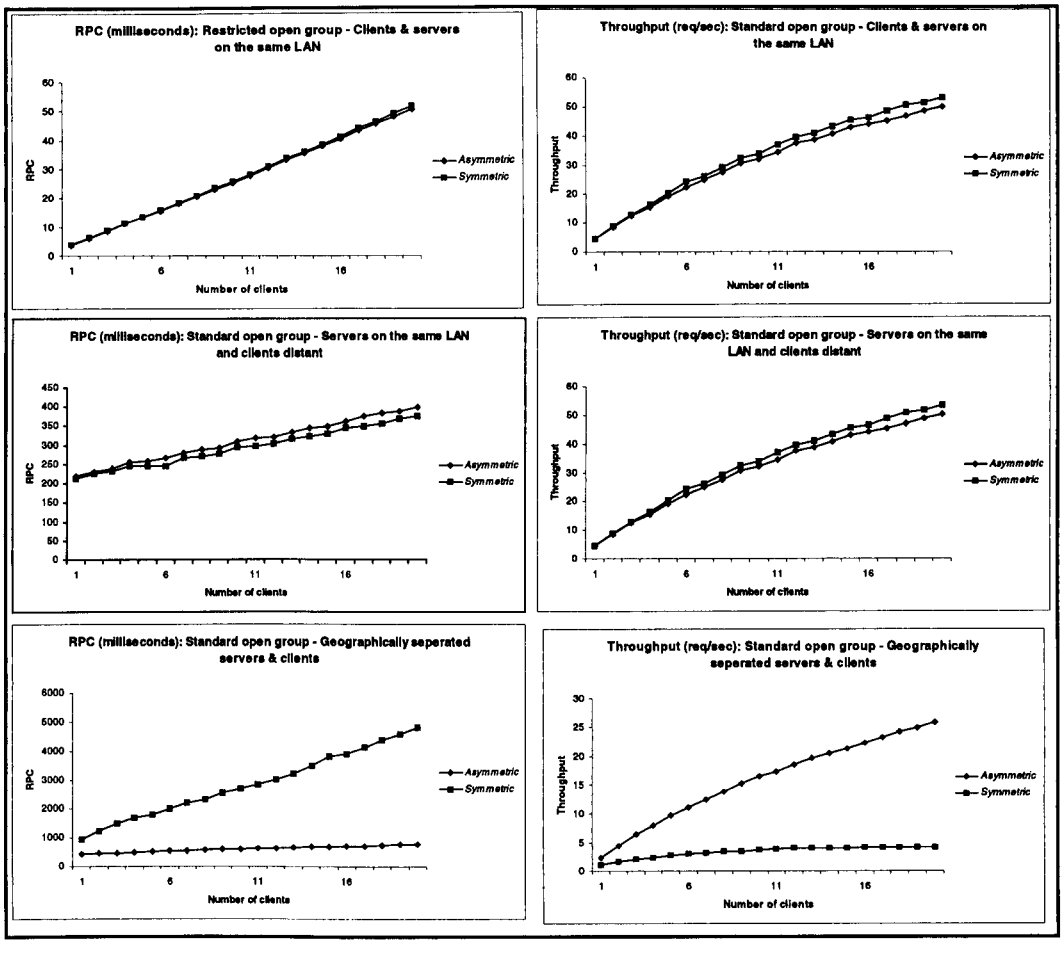

Figure 6.8 – Comparing asymmetric and symmetric protocols in the standard open group approach for request/reply experiments.

The performances of the symmetric and asymmetric protocols are very similar when servers are co-located. The symmetric protocol slightly outperforming the asymmetric protocol in both the LAN and the Internet 1 (co-located servers) experiments. The reason for their close correlation is due to the similar degrees of message volume and message blocking experienced by both protocols. The multicasting and message ordering associated to a client request is limited to a group of 3 members (the server group) whereas in the closed server group client involvement was required (a group of 4 members). The performance of a 3 member asymmetric group is still better than that of a 3 member symmetric group. Therefore, there is an expectation that the asymmetric should still outperform the symmetric. Unfortunately, due to the ability of any one of the servers to become request managers, message blocking will occur as client/server groups are synchronous and messages arriving at

different request managers may be causally related (see chapter 5). Furthermore, as any member of the server group may be a request manager the ability to use a shared sequencer is not an option. The message blocking encountered far outweighs any benefit gained from sequencer based ordering when the number of clients are relatively large compared to the size of the asymmetric server group as here. In practice the redirection of messages by a sequencer is more time consuming than simple multicast.

The Internet experiments are dominated by the slow Internet connection between clients and a server group. Such is this dominance that there is very little difference between a locally replicated server and a non-replicated server (fig. 6.4). This is because the only messages passing between a client and a server group are the RPC required to carry a request to a request manager and the RPC required to carry the replies from the request manager to the client. The same as a non-replicated server. Similar to the non-replicated server, as client numbers rise so the throughput rises. When servers are geographically separated then the asymmetric protocol does outperform the symmetric protocol. In the symmetric protocol message passing is required to ensure the ordering and reliability guarantees are satisfied. This is not the case in the presence of a shared sequencer as messages are deliverable as they are received. When servers are geographically distributed such message passing is time consuming, as message latency is high compared to that of co-located servers.

6.2.6 The Restricted Open group Approach

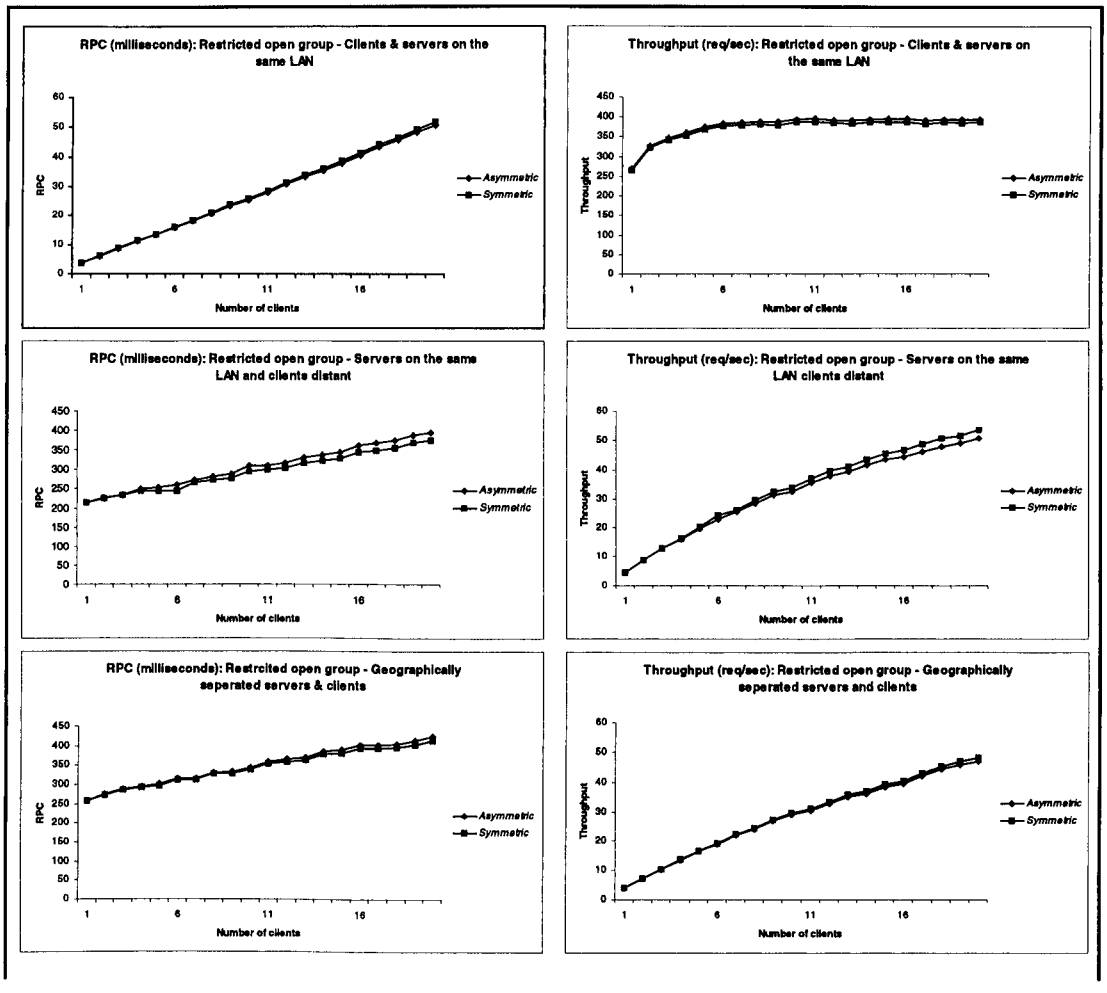

Figure 6.9 – Comparing asymmetric and symmetric protocols in the restricted open group approach for request/reply experiments.

In the LAN and Internet asymmetric experiments the sequencer chosen for the server group is the request manager. This produces identical message volume per request for both protocols; request manager receives client request, request manager asynchronously directs the request to other members of the group, request manager sends single reply. Furthermore, a client request may be serviced without the need for the asymmetric and symmetric protocols to experience message blocking or message passing for the sake of message ordering delivery guarantees (see 5.5.1). As message volume and message blocking is the same in both protocols there is an expectation that the

performance of both protocols would be the same, which is borne out by the results presented in fig. 6.9.

6.2.7 Peer Participation

In the peer participation experiments the symmetric ordering scheme is superior to the asymmetric ordering scheme. In the LAN environment the volume of messages that result from persistently sending asynchronous multicasts has resulted in a deterioration of performance in both the symmetric and asymmetric protocols as group membership rises. This deterioration is more extreme in the asymmetric protocol than the symmetric protocol. This indicates that the sequencer is receiving more messages than it can handle. The sequencer is a bottleneck. This bottleneck effect explains why the asymmetric performance deteriorates significantly as group membership rises. This deterioration is not evident in the symmetric scenario as the handling of messages and multicasting is more evenly spread throughout the group. In the Internet scenario, due to the large message transit times involved, the sequencer does not present a bottleneck. However, the cost of redirection is evident; the performance of the asymmetric protocol is approximately half that of the symmetric protocol. The results of the peer participation experiments are shown in fig. 6.10.

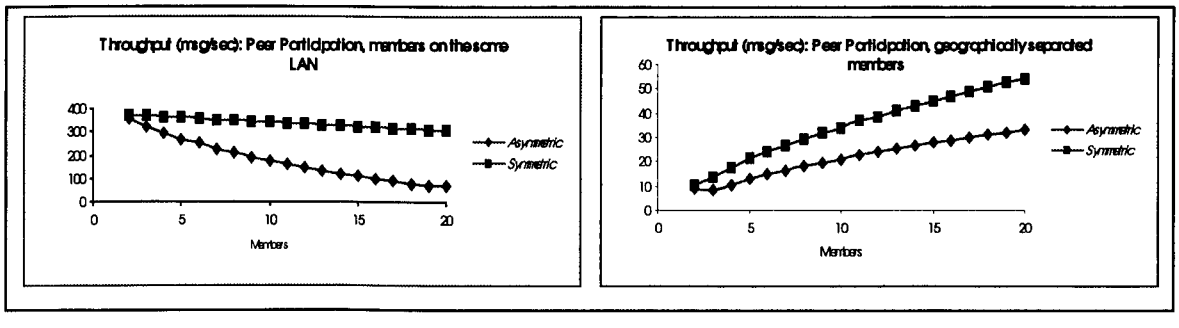

Figure 6.10 – Comparing asymmetric and symmetric protocols in peer groups.

To satisfy a synchronous request two messages must become deliverable; the initial request must become deliverable in the server group and the reply of the servers must become deliverable at the client. Considering this, there is an expectation that the throughput of the peer group should be twice that of the server group as the time taken for an asynchronous request to become deliverable is measured (in effect, the time taken for a request to become deliverable at the server only). This is not the case. The performance of the symmetric peer group is approximately the same as that of the highest performing request/reply group (restricted open) in both the LAN and WAN experiments. This can be explained by the frequency that members are sending asynchronous requests in the peer

group approach. The expense of persistently multicasting asynchronous requests is costly with regard to the processing resources available. Therefore, the protocol layer spends approximately the same amount of time sending multicasts as receiving them.

6.3 Summary

This chapter described a set of experiments involving the NewTOP service and presented the results from these experiments. These experiments were devised to determine the performance of protocol configurations that may be appropriate in the support of highly available and Groupware applications. The performance of the protocol configurations were measured as the throughput of client requests per second in a group. From the analysis of performances gained from the experiments presented in this chapter the following may be stated:

- The closed group approach provides the poorest performance of all request/reply approaches.
- Restricted open group provides the better performance of all the request reply scenarios. However, if the request manager fails (e.g., primary in passive replication) then a client may spend some time rebinding to another server.
- When clients are geographically distant from a server group, and the server group is located within the same LAN, there is little difference between the standard open group approach and the restricted open group approach. Furthermore, the cost of replication/group communication is negligible when Internet latency dominates.
- When clients and members of a server group are geographically separated the restricted open group approach significantly outperforms the standard open group approach.
- Asymmetric protocols are more appropriate for request/reply approaches, whereas symmetric protocols are more suitable to peer group approaches.

Chapter 7

Conclusions

This chapter will summarize the material which has been covered in the Thesis and give an indication of the possible areas of future research.

7.1 Thesis Summary

In the Thesis, main issues concerning the design and implementation of a fault-tolerant group communication service suitable for use in object-oriented Middleware environments have been presented. Such a service eases the development of applications that require group communications. There are a number of different types of applications that may benefit from a group communication service (e.g., highly available services, Groupware). Each type of application's group communication requirements may be different. For maximum generality, an asynchronous environment has been assumed. Therefore, it is possible that message transmission times can not be accurately estimated and the network may become partitioned. The Thesis contributes to the area by presenting a comprehensive approach to the provision of group communications suitable for satisfying the requirements of a wide variety of application types within object-oriented Middleware environments. The NewTOP service described here has been fully implemented.

7.1.1 A CORBA Service

An approach for providing a group communication service within the CORBA domain using only mechanisms as defined by the CORBA standard is presented. This ensures that such a service places no dependencies on underlying platforms or non-standard enhancements to CORBA. This approach (commonly termed the service approach) has the benefit of ensuring that the interoperability and portability of an application built using the service is preserved. Other approaches to service creation have been proposed:

- *Integration* - Group communication functionality is integrated into the ORB.
- *Interception* - IIOP messages are intercepted and mapped onto an underlying group communication sub-system.

These other approaches do not present the same interoperability and portability qualities as the service approach. The OGS also advocates the service approach (see 2.7.4). However, the OGS is not

flexible enough to support a wide variety of application types as it does not support overlapping groups. Furthermore, OGS does not support a choice between asymmetric and symmetric type ordering protocols.

7.1.2 Group Communication Protocols

Symmetric and asymmetric message ordering protocols, capable of handling dynamic groups have been presented. These protocols were originally designed to support total ordered message delivery only. In an attempt to satisfy a wider range of application requirements, these protocols have been extended to provide an application developer with a choice between causal ordering, arbitrary ordering and total ordering on a per group basis. Furthermore, additional functionality has been added to the causal and total ordering protocols to reduce the necessity of message blocking and therefore potentially increase message throughput within overlapping asymmetric groups that share the same sequencer. As with message ordering, the developer may choose when to employ such optimization techniques. Although related works do allow overlapping groups (see 3.6) they do not allow members that simultaneously participate in multiple groups to use say, symmetric ordering protocols in one group while using asymmetric ordering protocols in another. To the best of my knowledge there are no other related works that provide application developers with a choice of functionality that has the potential for reducing message latency within causal and total ordering protocols via overlapping groups.

A failure suspector based on the causal relationships between messages and timeouts has been developed. Based on this failure suspector, a membership protocol has been developed that ensures network partitions do not lead to group members forming inconsistent views of the membership of a group. Message delivery is kept atomic with respect to view change installations.

7.1.3 Interactions Between Clients and Server Groups

Protocols for open and closed client/server group interactions have been developed. Such protocols enable clients to interact with server groups via the overlapping of groups. Two types of groups are identified; client/server and server groups. Clients participate in client/server groups with one (open group approach) or more servers (closed group approach). In principle, clients issue requests and receive replies from a server group g_X by participating in a client/server group that overlaps with the server group g_X.

For open groups, a mechanism has been presented that enables a client to issue requests to all members of a server group and receive server replies via a single member of the server group (request manager). An application developer may identify an open server group as restricted. Restricted server groups only have one request manager. The existence of only one request manager in a server group

ensures a single point of entry into the server group for client requests. Such a scenario reduces the need for message blocking in the underlying group communication protocols when ensuring the ordering and delivery guarantees of messages associated with client requests.

In both approaches (open and closed groups) the number of replies a client waits for as a result of a request may be specified on a per request basis. Furthermore, the manner in which such replies are handled when members of the server group fail during the processing of a client request may be specified by an application developer.

7.1.4 Performance of the NewTOP Service

The implementation of the NewTOP Object Group Service has been described. Experiments have been run to evaluate the service in two different network environments; (i) all objects restricted to a LAN, (ii) objects separated geographical by large distances enabling inter-object communication via the internet.

Experiments were performed and data was gathered that aided in the comparison of the performance of symmetric and asymmetric total ordering protocols under different types of group strategies (client/server groups, peer groups). As a brief summary; in client/server group type interactions (where it is common for only a single member to multicast while other members are idle) the asymmetric protocol outperforms the symmetric protocol, whereas, in peer group (where it is common for all members to be frequently multicasting) the symmetric protocol outperforms the asymmetric protocol. An advantage of the NewTOP service over other group communication services is that the NewTOP service provides both asymmetric and symmetric ordering, the choice of which is left to the application developer.

7.2 Main Contributions

The contributions made by this Thesis can be summarized as follows:

i. A service suitable for satisfying the group communication requirements of a wide variety of application types has been presented within an object-oriented Middleware environment (CORBA). There has been no need to enhance CORBA in a non-standard way, nor has any reliance been placed on underlying network technologies. This ensures that applications built using the service retain the interoperability and portability associated with CORBA.

ii. The Thesis has discussed and developed group communication protocols. These protocols enable overlapping groups while retaining low message overheads. Asymmetric and symmetric message ordering protocols are presented that may be used on a per group basis,

permitting groups that are asymmetric to overlap with groups that are symmetric. Furthermore, these protocols may be tailored to suit a wide variety of application types.

iii. Client/server group interactions of an open and closed type have been addressed within the group communication protocols via overlapping groups. Furthermore, the problems relating to message repetition when a client is itself a group have been addressed.

iv. The work presented here has been implemented as a CORBA service and an array of experiments have been carried out on it. The analysis of these experiments suggest the overlapping group and configurability strategies available for application developers do allow the service to be tailored to suit applications of quite different types in various network environments.

7.3 Future Work

The work presented here can be extended in several directions:

7.3.1 Merging Groups that are a Result of a Partition

The membership protocol presented in the Thesis is capable of dealing with network and virtual partitions. A partition leads to situations where a group is divided into sub-groups of functioning objects. These sub-groups have no knowledge of each other, are completely disconnected and share the same group identifier. The membership protocol presented here is designed so that message delivery will be kept atomic with respect to view installations in each of the partitioned sub-groups. However, the problem of merging sub-groups holding the same group identifier, which could happen, for instance, after a partitioned network has been "fixed" is not addressed by the Thesis.

This problem has been addressed for different sets of minimal repair and recovery conditions [Ezhilchelvan99, Lotem97]. We intend to integrate [Ezhilchelvan99] with the NewTOP service in the near future.

7.3.2 Economical Asynchronous Communications

The NewTOP service accomplishes a multicast by directing synchronous RPCs at each member of a group. This is made asynchronous (required to ensure correct working of the group communication protocols) by assigning each multicast request to its own thread of control. Using a synchronous RPC to enable the inter-object communication is costly. This is because a synchronous RPC results in a redundant message; the message sent from the server to the client carrying server replies. Furthermore, the ORB will marshal and unmarshal the contents of an RPC even though such functionality is provided at the invocation layer of the NewTOP service. To overcome the waste of network and process resources, that are an inevitable result of redundant message passing and

marshaling/unmarshaling functions, and to allow application developers to use other transport protocols apart from IIOP the latest CORBA standard (CORBA3) provides:

- *Messaging service* [OMG98b] - A CORBA service that provides application developers with asynchronous message passing.
- *Alternate transport protocols* - Application developers may introduce their own transport level protocols to the CORBA environment.

The NewTOP service can be refined to exploit these new facilities.

7.3.3 Replication Support for Transactional Objects

Atomic transactions ensure that only consistent state changes take place to objects despite concurrent access and failures. However, they may be insufficient to ensure that an application makes forward progress. Although data replication techniques using transactions are well known, in distributed object environments, facilities for server replication as provided by group communications can provide additional flexibility [Little99].

Given that CORBA supports a transaction service (Object Transaction Service - OTS), work on using a group communication service and OTS together to support a highly available transactional approach to distributed application design would be very interesting.

References

[Agarwal94] D. A. Agarwal, "Totem: A Reliable Ordered Delivery Protocol for Inter-connected Local Area Networks", PhD Thesis, Department of Electrical and Computing Engineering, University of California, Santa Barbara, 1994.

[Amir92] Y. Amir et al, "Transis: A Communication Sub-system for High Availability", Digest of Papers, FTCS-22, p 76-84, 1992

[Babauglu94] O. Babaoglu et al, "On Group Communications in Large-Scale Distributed Systems", Proceedings of the 6th ACM SIGOPS European Workshop, pages 17 - 22, Germany, 1994.

[Babaoglu94] O. Babaoglu et al, "Relacs: a communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems", BROADCAST Project deliverable report (available from Dept. of Computing Science, University of Newcastle upon Tyne, UK).

[Bal90] H. E. Bal, " Programming Distributed Systems", Silicon Press, 1990

[Barrett90] P. A. Barrett et al, "The Delta-4 Extra Performance Architecture (XPA)", Proceedings of FTCS-20, Newcastle upon Tyne, June 1990.

[Bernstein96] P. A. Bernstein, "Middleware: A Model for Distributed System Services", Communications of the ACM, Vol. 39, No. 2, pages 86 - 98, 1996.

[Birman87] K. Birman et al, "Exploiting Virtual Synchrony in Distributed Systems", Proceedings of the 11th Symposium on Operating Systems Principles, 1987.

[Birman91] K. Birman et al, "Design Alternatives for Process Group Membership and Multicast", Technical Report TR91-1257, Department of Computing Science, Cornell University, December 1991.

[Birman93] K. P. Birman, "The Process Group Approach To Reliable Distributed Computing", Communications of the ACM, 36(12), pages 37 - 52, 1993.

[Birrell84] A. D. Birrel et al, "Implementing Remote Procedure Calls", ACM Transactions on Computing Systems, vol. 2, pages 39 - 59, 1984.

[Brown96] N. Brown et al, "Distributed Component Object Model Protocol - DCOM/1.0" Microsoft Inc., 1996.

[Chandra91] T. D. Chandra et al, "Unreliable Failure Detectors for Asynchronous Systems", Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, p 325-340, ACM, August 1991.

[Cristian91a] F. Cristian, "Understanding Fault-Tolerant Distributed Systems", Communications of the ACM, 34(2), February 1991.

[Cristian91b] F. Cristian, "Reaching Agreement On Processor Group Membership In Synchronous Distributed Systems", Distributed Computing, 4(4), pages 175 - 187, 1991.

[Cristian96] F. Cristian, "Synchronous and Asynchronous Group Communications", Communications of The ACM, 39(4), pages 88 - 97, 1996.

[Dolev87] D. Dolev et al, "On the Minimal Synchronisation Needed For Distributed Consensus", Journal of the ACM, 34(1), pages 77 - 97, January 1987.

[Dolev93] D. Dolev et al, "Early Delivery Totally Ordered Multicast in Asynchronous Environment", Digest of Papers, FTCS-23, Toulouse, p544-553, 1993.

[Dwork88] C. Dwork et al, "Consensus in The Presence of Partial Synchrony", Journal of the ACM, 35(2), pages 288 - 323, April 1988.

[Ezhilchelvan86] P. Ezhilchelvan and S.K. Shrivastava, A characterization of faults in systems. Proc. of 5th IEEE Symp. on Reliability in Distributed Software and Database Systems, Los Angeles, pp. 215-222, January 1986.

[Ezhilchelvan95] P. Ezhilchelvan95 et al, "Newtop: a fault-tolerant group communication protocol", 15th IEEE Intl. Conf. on Distributed Computing Systems, Vancouver, pp. 296-306, May 1995.

[Ezhilchelvan99] P. Ezhilchelvan and S K Shrivastava, "Enhancing Replica Management Services to Tolerate Group Failures", Proceedings of the second International Symposium on Object oriented Real-time Computing (ISORC), May 1999, St Malo, France

[Felber98a] P. Felber, R. Guerraoui and A. Schiper, "The implementation of a CORBA object group service", Theory and Practice of Object Systems, 4(2), 1998, pp. 93-105.

[Felber98b] P. Felber, "The CORBA Object Group Service: a Service Approach to Object Groups in CORBA", PhD thesis, Ecole Polytechnique Federale de Lausanne, 1998.

[Fischer85] M. Fischer, "The Consensus Problem in Unreliable Distributed Systems", Proceedings of the International Conference on Foundations of Computing Theory, Sweden, 1983.

[Garcia-Molina91] H. Garcia-Molina, "Ordered and Reliable Multicast Communication", ACM Transactions on Computing Systems, Vol. 9, No. 3, p 242-271, August 1991

[Grand98] M. Grand, "Patterns in Java, Volume 1", Wiley, 1998.

[Hadzilacos93] J. Y. Hadzilzcos, S. Toueg, "Fault-Tolerant Broadcasts and Related Problems", Chapter 5, pages 97-145, ACM Press Frontier, Addison-Wesley, 2nd edition, 1993.

[Iona94] Iona Technologies Ltd. And Isis Distributed Systems, "An Introduction to Orbix+Isis", 1994.

[Iona95] Iona Technologies Ltd. "Orbix 2 Programming Guide", 1995.

[Karamanolis96] C. T. Karamanolis, "Configurable Highly Available Distributed Services", PhD Thesis, Department of Computing, Imperial College of Science, Technology and Medicine, 1998.

[Karamanolis99] C. T. Karamanolis et al, "Client Access Protocols for Replicated Services", IEEE Transactions on Software Engineering, Vol. 25, No. 1, p3-22, January, February 1999.

[Kennington96] S. Kenington, "Netscape Communicator 4.0 Made Simple", Made Simple, 1996.

[Kramer91] S. Kramer et al, "Transis: A Communication Sub-System for High Availability", Technical Report TR 91-13, Computer Science Department, The Hebrew University of Jerusalem, 1991.

[Lamport78] L. Lamport, "Time Clocks and The Ordering Of Events in a Disrtibuted System", Communications of The ACM, 21(7), Pages 558 - 565, 1978.

[Lamport82] L. Lamport et al, "The Byzantine Generals Problem", ACM Transactions of Prog. Lang. And Systems 4, 3, pages 382-401, 1982.

[Little92] M. C. Little, "Object Replication in a Distributed System", PhD Thesis, University of Newcastle upon Tyne, 1992.

[Little99] M.C. Little, S.K. Shrivastava, "Implementing high availability CORBA applications with Java", Proceedings of the IEEE Workshop on Internet Applications, San Jose, California, June 1999.

[Lotem97] E. Y. Lotem, I. Keidar and D. Dolev, "Dynamic Voting for Consistent Primary Components", Proceedings of ACM Symposium on Principles of Distributed Computing (PODC), pp. 63-71, 1997.

[May84] D. May et al, "The Transputer Implementation of occam", Proc. Int. Conf. On Fifth Generation Computer Systems, p 533-541, November 1984

[Macedo95] R. J. De A. Macedo "Fault-Tolerant Group Communication Protocols For Asynchronous Systems", PhD Thesis, Department of Computing Science, University of Newcastle upon Tyne, 1995.

[Maffeis95] S. Maffeis, "Run-time Support for Object-Oriented Distributed Programming", PhD Thesis, University of Zurich (Switzerland), 1995.

[Melliar-Smith91] P. M. Melliar-Smith, "Membership Algorithms for Asynchronous Distributed Systems", Proc. Of 12th Intl. Conf. On Distributed Comp. Systems, p 480-488, 1991

[Melliar-Smith94] P. M. Mellier-Smith et al, "Extended Virtual Synchrony", Proceedings of the 14th IEEE International Conference on Distributed Computing Systems, Poland, pages 56 - 65, 1994.

[Melliar-Smith97] P. M. Mellier-Smith et al, "Replica Consistency of CORBA Objects in Partitionable Distributed Systems", Distributed Systems Engineering 4, p 139-150, 1997

[Mishra91] S. Mishra et al, "Consul: A Communications Substrate for Fault-Tolerant Distributed Programs", Distributed Systems Engineering, p 87-103, 1993.

[Morgan99] G. Morgan et al, "Design and implementation of a CORBA fault-tolerant object group service", Proceedings of the conference on Distributed Applications and Interoperable Systems, 1999.

[Morin98] T. Morin, "Migrating Legacy Systems To CORBA", Object Magazine, "http://www.sigs.com", p39-43, January 1998.

[Narasimhan97] P. Narasimhan et al, "Replica Consistency of CORBA Objects in Partitionable Distributed Systems", Distributed Systems Engineering (4), pages 139 - 150, 1997.

[OMG95a] The Object Management Group, "The Common Object Request Broker Architecture and Specification", 1995.

[OMG95b] The Object Management Group, "Interface Repository", Framingham MA, version 1.0.2 edition, OMG TC Document 95-1-147, 1995.

[OMG98a] The Object Management Group, "Fault Tolerant CORBA Using Entity Redundancy Request For Proposal", OMG document: orbos/98-04-01, "http://www.omg.org", April, 1998.

[OMG98b] The Object Management Group, "CORBA Messaging", OMG document: orbos/98-05-05, "http://www.omg.org", May, 1998.

[OOC98] Object Oriented Concepts Inc. "http://www.orbacus.com".

[O'Malley89] S. O'Malley et al, " RPC in the x-kernel: Evaluating New Design Techniques", Proceedings of the 12th Symposium on Operating Systems Principles, Litchfield Park, Arizona, 1989.

[Ranka88] S. Ranka et al, "Programming a Hypercube Multicomputer", IEEE Software, Vol. 5, No. 5, p 66-77, September 1988

[Renesse96] R. van Renesse et al, "Horus, a flexible Group Communication System", Communications of the ACM, 1996.

[Ricciardi91] A. M. Ricciardi et al, "Using Process Groups to Implement Failure Detection in Asynchronous Environements", Proc. Of Annual ACM Symposium on PoDC, p 341-352, 1991.

[Ritchie84] D. M. Ritchie, "A Stream Input-Output System", Bell Laboratories Technical Report, Journal 63, 1984.

[Rosenberry92] W. Rosenberry et al, "Understanding DCE, OSF Distributed Computing Environment", Addison Wesley, 1992.

[Savets96] K. Savets, "MBONE", IDG Books World Wide, 1996.

[Schiper93] A. Schiper et al, "Virtual Sunchronous Communication Based on a Weak Failure Suspector", Digest of Papers, FTCS-23, Toulouse, p 534-543, 1993.

[Schlichting83] R. D. Schlichting, F. B. Schneider, "Fail-Stop Processors: An Approach To Designing Fault-Tolerant Computing Systems", ACM Transactions on Computer Systems, August 1983.

[Schmidt93] D. C. Schmidt, "ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integration and Evaluation Environment", Concurrency: Practice and Experience, vol. 5, no. 4, pages 269 - 286, 1993 .

[Shrivastava90] S. K. Shrivastava et al, "Fail-Controlled Processor Architectures For Distributed Systems", Technical Report, Department of Computing, University of Newcastle upon Tyne, 1990 .

[Shrivastava91a] S. K. Shrivastava et al, "A Overview of the Arjuna Distributed Programming System", IEEE Software, 1991.

[Shrivastava91b] S.K. Shrivastava, "Fault-tolerant system structuring concepts", Software Engineer's Reference Book (ed. J. McDermid), Chapter 61, Butterworth-Heinemann, 1991.

[Soley95] R. M. Soley et al, "The Object Management Architecture Guide", John Wiley & Sons Inc., third edition, 1995.

[Sloman87] M. Sloman et al, "Distributed Systems and Computer Networks", Prentice Hall, 1987

[Stevens98] W. R. Stevens, "TCP/IP Illustrated Volume 1 : The Protocols", Addison Wesley, 1998.

[Stevens99] W. R. Stevens, "UNIX Network Programming, Network APIs : Sockets and XTI", Prentice Hall, 1999.

[Sun97] Sun Microsystems, "Java Remote Method Invocation Specification", JDK 1.1, February 1997.

[Tanenbaum85] A. S. Tanenbaum et al, "Distributed Operating Systems", ACM Computing Surveys, Vol. 17, No. 4, pages 419-470, December 1985