Object Groups For Groupware Applications: Application Requirements and Design Issues

Richard Achmatowicz

richarda@dcs.qmw.ac.uk
Queen Mary and Westfield College, Dept. Of Computer Science
Mile End Road, London E1 4NS, United Kingdom

Technical Report No. 685

September 1994

Abstract

Process groups have proven successful in structuring directly distributed, process-oriented applications in a wide variety of application areas. We consider here the suitability of an object group construct for constructing distributed, object-oriented applications from the application domain of real-time groupware. Application requirements of groupware are considered. Design issues for object groups are then discussed in light of the application requirements. Several existing object group systems are examined and evaluated in terms of their suitability for meeting the requirements of groupware applications. Finally, a summary of the important design issues is presented.

Keywords and Phrases: Object-oriented Programming, Groupware, Object Groups, Object Semantics

1 Introduction

Groupware is software designed to allow groups of people to use computers to work together closely to achieve a common task. It places a unique set of requirements on the application and system designer.

One class of groupware applications are *shared document editors.* A shared document editor allows multiple users residing at different workstations in a local area network to concurrently edit a single document. The application needs to manage concurrent changes to the document in such as way as to allow non-conflicting operations to succeed and warn users when operations conflict. It must provide functionality to allow users to productively collaborate in the preparation of complex documents.

Another class of groupware applications are *multi-user virtual reality* applications, which allow one or more users to move through a virtual world. The world represents a common or shared environment within which the users navigate. As the environment is shared, the effect of one user on the world generally needs to be observable by the other.

Groupware also encompasses application classes such as distributed conferencing systems, hypertext document management systems, and collaborative design tools. A good summary of the scope of groupware applications can be found in [Ellis91].

In this report, we shall examine the unique requirements which distributed, object-oriented groupware applications place on the application designer, and shall examine how *object groups* [Shapiro91, Hagsand91, Maffeis93, ANSA93b, ISIS94] might form the basis of an architecture for structuring groupware applications. An object group is a collection of objects which can be treated as a single logical entity. The object group abstraction has several characteristics which make it attractive for structuring distributed groupware applications.

The rest of this section shall present the characteristics of real-time, distributed groupware applications and introduce the object group abstraction, providing a basis for the discussion. A summary of the structure of the rest of the report is provided at the end of this section.

1.1 Characteristics of Groupware Applications

Following [Ellis89], we first define some terminology. An execution of a groupware system is called a *session*, with the users referred to as *participants* in the session. Each participant has a user interface to the *shared context* maintained by the groupware. The system's *response time* is the time required for a user's actions to be reflected in their own user interface, and the *notification time* is the time necessary for one user's actions to be propagated to the user interfaces of the other users.

Some characteristics of real-time, object-oriented groupware which are of interest for the purposes of this report are as follows:

Distributed. In general, the participants of a groupware session will not be connected to the same machine, and, although they may each be using machines connected by the same local or wide area network, may be in different physical locations. For example, it should be possible for two users to edit a shared document even if in physically disparate locations, such as London and Edinburgh.

Shared Environment. Groupware systems involve allowing participants access to a shared environment - a shared document in the case of a shared editor, or a shared virtual world in the case of a multi-user virtual reality application.

Highly Interactive. Real-time groupware systems are highly interactive and require fast response times. The user of a shared editor will not want to be delayed in carrying out editing tasks, as he expects response times similar to a single user editor. Similarly, to provide a proper feeling of presence in a virtual world, changes to the world resulting from user actions must be comparable to those experienced in the real world.

Closely Coupled With Other Participants. Groupware is unique in that information about other participants is required in many cases for successful group work to take place. In this sense, the applications making up the groupware system have to be closely coupled to each other, and not isolated from each other as in traditional data-oriented applications. Participants are often given mechanisms for communicating with other participants in the session.

Real-Time Notification Of Events. Because of the close coupling of users in a groupware application, we need real-time notification of the events generated by other participants. In the case of a shared editor, this could refer to updates to a shared document, or a participant leaving a session. In the virtual world, we would want to know if someone was entering the room we were in.

Many of these requirements are conflicting and make application support for groupware difficult. For example, the fact that groupware applications are distributed makes it difficult to provide access to a shared environment and, at the same time, maintain both fast response time and notification time.

Further, even though groupware systems are multi-user systems, traditional multi-user systems have focused on separating users from each other. Multi-user applications which require close coupling of user interactions present a new set of challenges for traditional application architectures.

Groupware application architectures generally fall into one of two classes [Lauwers90, Greenberg92]: *centralized* or *replicated.* In the centralized architecture, the application is composed of a single process, the central agent, and one participant process for each user workstation participating in the session. The participant processes merely accept user input and display output on the local workstations, passing all information to the central agent which manages the application state. In the replicated architecture, the application program is replicated on each of the participant's user workstations and participant processes communicate directly with one another. They are collectively responsible for maintaining the state of the application.

[Greenberg92] has noted advantages and disadvantages of each architecture. In particular, centralized applications make management of shared data easier as all data is in one location, at the central agent. Synchronization and concurrent access to data can thus be managed using well-known database techniques. On the other hand, a central agent is a processing bottleneck and permits a single point of failure. Replicated architectures avoid single points of failure through replication of processing agents, but make the management of shared data and synchronization of the various agents difficult.

In this report, we shall be concerned solely with replicated architectures and the problems associated with constructing groupware applications based on replicated architectures.

1.2 Object Groups

An *object group* is a collection of objects which can be treated as a single logical entity for the purposes of invocation. An invocation made on the object group itself results in invocations on one or more of the objects making up the object group. There has been only a limited amount of research into the object group abstraction, and no single definition has emerged. However, we shall assume that object groups have the following characteristics:

Named Collection. An object group is a *named collection* of objects with mechanisms allowing objects to join and leave the object group. The objects making up the collection are referred to as group *members*. Objects in the system which make invocations on an object group are referred to as *clients* of the object group ;

Interface. An object group presents an interface in the form of a collection of methods which the group supports, called its *group interface* ;

Communication. A form of *group invocation* is provided which allows objects to communicate with group members without necessarily knowing who the members of the group are.

Object groups give rise to new invocation semantics which apply to the collection as a whole and are known as *group invocation semantics.*

An *object group system* is a level of system software which directly supports the object group abstraction. Object groups systems can provide varying degrees of support for programming with object groups. There are several object group systems in existence at the time of writing. Some are commercially available, some only existing as research prototypes. They are largely differentiated by the programming model they provide, each with a different emphasis on how to use the object group abstraction to structure applications.

In a distributed system, the processes making up an object-oriented application can be executing on different machines in the network. In such a case, objects need to be able to make invocations on other *remote objects* outside their own address space. This requires a form of *remote method invocation*, where an object on one machine makes an invocation on a named object on another machine. We generally assume that the objects making up the object group may be located on different machines in the network, although is possible (and useful) to group objects on the same machine together.

Classifying Object Groups

There are several types of object groups which have appeared in the literature. They can be differentiated by one or more of several criteria: the type and function of their member objects, the properties of the communication they provide to members and clients of the group and the level of information members have about each other. We now define some of the terminology used to describe the various types of object groups:

Open and Closed. If a non-member of the group can make an invocation on the group interface, the group is said to be *open.* If communication with the group requires membership in the group, the group is said to be *closed.* Closed groups are also referred to as *peer* groups.

Homogeneous and heterogeneous. An object group is said to be *homogeneous* if all its members are instances of the same object class; otherwise, the group is said to be *heterogeneous.*

Active and Passive. An object group is said to be *active* if each invocation of the group interface results in an invocation on the interfaces of the groups members. A *passive* object group is one in which a group invocation is passed to a single object group member: the other members are passive observers of the activity of the single, distinguished member. This term usually applies only to homogenous groups where one group member performs the work of the group and others checkpoint its state every once in a while.

Transparent and Non-transparent. An object group is *transparent* if the members of the group do not have information about the other group members. *Non-transparent* groups are those in which each object group member has certain information about the current set of group members; this is usually at least a list of identifiers which can be used to communicate with the other members by sending individual messages or using the group communication mechanism. Transparent groups are also referred to as *anonymous* groups; non-transparent groups are sometimes referred to as *explicit* groups [Birman89].

Replica. Replica groups are homogeneous groups where the group invocation semantics are the same as the invocation semantics for a single object of the member class. Replica groups may be active or passive.

Process Groups

Object groups are an object-oriented counterpart to *process groups* which have been studied extensively in the literature [Birman89, Birman93a, Kaashoek91, Cheriton86]. Process groups form the basis of the process group application architecture for distributed applications, and exhibit significant differences from the processor pool architecture and the client-server architecture [Coulouris94].

The process group application architecture views a distributed application as consisting of processes, process groups, and broadcast events. Broadcast events encompass group communication between processes, point-to-point communication between processes, and process failure events. Process groups are well-suited to structuring distributed applications which require close cooperation between application components.

Following [Birman89], we make the following distinction between classes of distributed applications:

Directly Distributed Applications. Applications in which distributed threads of execution interact directly with one another while continuously respecting constraints on their joint behaviour.

Data Oriented Systems. Distributed threads of execution interact indirectly with one another by issuing operations on shared data objects.

From these definitions, we can see that the replicated groupware architecture described previously is an example of a directly distributed application, whereas a groupware application based on a central agent process managing shared data can be classed as a data oriented system. Examples of groupware based on a replicated architecture include GROVE [Ellis91], GroupSketch and GroupDraw [Greenberg91, Greenberg92], DistEdit [Knister90] and DIVE [Fahlen93, DIVE93]. Groupware based on a centralized architecture include Ensemble [Newman92], DeVise [Gronbaek94], and IMP [Bates94].

Process groups are well-suited to structuring directly distributed applications as they provide the required group membership information and group communication primitives. They have also been used to support implementations of some of the important techniques used in distributed applications, such as replication, fault-tolerance and parallel processing [Birman89, Birman93a].

Replication is a general technique that can be used to bring benefits to several aspects of distributed systems. In general, we can either replicate data, or we can replicate execution. There are several types of replication used in distributed applications:

> *Replication of Data.* Replication of data structures used by processes can aid in parallel programming on loosely coupled architectures. The replicated data is used to simulate the traditional parallel programming paradigm of shared memory [Bal88].
>
> *Replication of Execution.* Replication of execution can be used as in a load-balancing scheme when a service is represented by a collection of replicated servers, each able to process service requests independently.
>
> *Replication for Fault-tolerance.* Replication can be used to provide fault-tolerance, either through replicated execution or through a primary-backup technique.

As mentioned earlier, object groups are an object-oriented counterpart to process groups. We expect to gain all the benefits from a object-oriented programming model based on object groups as are being derived from the process groups programming model.

1.3 Objectives

We believe object groups provide a promising approach to structuring groupware applications based on a replicated architecture. Firstly, there are structural similarities: both groupware applications and object groups involve the group abstraction and associated group operations, such as joining and leaving groups, sending messages to group members, and managing the state of a group through direct cooperation. Secondly, groupware has, as we shall demonstrate, certain application requirements which are well-suited to an implementation in terms of object groups, such as various forms of event notification and access to a pool of data shared between all group members.

In this report, we aim to examine more closely the requirements of distributed groupware applications and look at how these requirements would affect the design of an object groups system intended to make writing groupware applications easier.

Structure of the Report

The structure of this report is as follows. In Section 2, we shall consider the requirements that the distributed and collaborative nature of groupware applications place on the groupware application designer. These requirements will need to be satisfied by any object groups system which is to be used to structure groupware applications. Section 3 will provide an examination of the design issues for object groups which arise from these application requirements. In Section 4, we shall examine several existing object groups systems to see how they match up to these design requirements. Finally, we shall summarise the main design issues for object groups which need to be addressed in order for them to provide a suitable approach to structuring groupware applications.

2 Requirements Of Groupware Applications

The characteristics of groupware applications described earlier lead to a set of requirements which set them apart from single-user distributed applications and which are particularly relevant to potential benefits of object groups. In this section, we shall attempt to identify those requirements.

2.1 Group Abstraction

Objects need the ability to join and leave a group for membership purposes. For example, the registration of new participants in a groupware session can be modelled as joining a group representing the session with the new group member receiving a copy of the group's current state [Greenberg92]. In certain application domains, subsets of objects are required to implement desired behaviour. In the Interactive Multimedia Presentation environment [Bates93], video and audio data streams and contexts(composite objects) are grouped into *context groups* which allow control of the individual contexts (stopping, starting, synchronizing, pausing, etc.) to be applied to the group as a whole. In the Collaborative Editing System [Greif87], a shared document object is made up of section objects which may reside on different hosts but which are treated as a single collection.

Together with the group abstraction goes the requirement of being able to make invocations on the individual group members by making a single invocation on the group. Due to the distributed nature of most groupware applications, group invocation must apply to remote objects as well as local ones. Various researchers [Ellis89, Greenberg92] have noted that messages delivered to the group need to satisfy certain ordering constraints if the shared state of the application is to be preserved, in the case where the application architecture is distributed as opposed to centralized.

2.2 Access To Shared Data

Perhaps the single most important characteristic of groupware applications is the concurrent access to the shared data of the application by the participants. Shared data provides the means for users to collaborate as a group on a single task. In collaborative text editing and collaborative drawing tools [Greif87, Greenberg92], each participant sees the entire document or drawing being edited on the screen and can see changes being made by other participants as they occur. Software designers can access a shared body of application code in development using the Revision Control System [Tichy82]. In a multi-user virtual reality system, users interact with each other through the shared virtual environment.

In a replicated groupware architecture, where participant processes may be geographically dispersed and subject to high network latencies, shared data needs to be replicated at all participant processes for fast local access. Delays in response time due to accesses to remote shared data would interrupt the participants use of the groupware application and possibly compromise the result. But a replicated object which is shared between several cooperating processes will likely be subject to concurrent invocations by the processes in question. Such concurrent accesses could lead to violation of consistency requirements, not only of the shared data object but the application as a whole. Thus, the requirement of shared data between participant processes leads to a requirement for concurrency control mechanisms to preserve the consistency of that shared data.

There is also a requirement for indivisible or *atomic* actions on shared data in groupware. For example, in many groupware applications, when a new participant joins a groupware session (registration), existing participants in the session need to be notified of the new member, and the new member needs to receive a copy of the current application state. If these two operations are not performed indivisibly, updates to the state which depend on group membership could result in inconsistency. The use of atomic operations in constructing groupware applications has been investigated in the development of the Collaborative Editing System [Greif87].

Concurrent access to shared data has been thoroughly studied in the case where one user's actions are logically separate from the actions of another [Bacon92]. Many of these traditional methods of concurrency control, such as locks and transactions, are based on the concept of serializability as a model for guaranteeing consistency of shared data.

Unlike the more traditional applications where the activities of one user are logically separate from another, the actions of groupware participants on shared data need to be tightly coupled in order to achieve collaboration. Users also need to be aware each others actions. Long-lived transactions, lasting even for days, have been identified as required in computer-aided design systems where one design drawing may involve use of several others. This presents serious challenges for traditional concurrency control techniques. Further, it has been shown that the concurrency control mechanisms chosen have a direct effect on the properties of the user interface and hence the suitability of the application to group work [Greenberg94]. In the case of traditional locking and transaction mechanisms, delays incurred when obtaining locks to shared data items can interrupt the flow of work significantly. Non-deterministic reordering of operations due to the strict serializability properties of transactions can also be disturbing for the user. Good analyses of how traditional concurrency control techniques fail for groupware can be found in [Ellis88, Greenberg94].

In order to provide concurrency control mechanisms which avoid these problems, concurrency control techniques specifically applicable to groupware applications have been developed and include new forms of collaboration-aware locks and transactions, as well as transactions which relax the traditional notions of consistency for shared data. Several of these mechanisms will be described in Section 3.2.2 when considering the concurrency control mechanisms required for replicated shared data based on object groups.

An alternative approach to maintaining consistency of shared data is not to have any concurrency control at all. [Greenberg90] observed that the lack of concurrency control in GroupSketch did not affect its usefulness as a drawing tool. The GROVE editor [Ellis88] explored maintaining consistent application state in the absence of any concurrency control by constraining the order in which operations on shared data are applied at each participant workstation. The work on the object-oriented drawing package GroupDraw [Greenberg92] concludes that several mechanisms with differing characteristics should be made available to the programmer.

2.3 Event Notification

When participants in a group cooperate, it is important that each participant be aware of the actions of others on which they depend. For example, in human group communication, this awareness can be achieved through various means (gesturing, writing a letter, making a telephone call, shouting). In groupware applications, there is a strong requirement for actions or events triggered by one object to be communicated to one or more other objects.

For example, changes in an object's state may be of interest to other objects. In the VEOS [Coco93] system, objects in a shared virtual environment can register interest in a particular state variable of another object, and 'be notified' when that state variable has changed. Such an object interaction facility could be used to allow a virtual light object to inform other objects in a room that the virtual light is on. Similarly, the dVS virtual reality software environment [DIVISION93] allows objects to register interest in other objects and be notified when a collision occurs.

The announcing of an event can be associated with an action. This requirement comes in several guises. One form of this requirement is referred to as *implicit invocation.* An object registers interest in an event at an object as before, but instead of a general message being sent to all interested objects, a particular method associated with each interested object is invoked automatically. The method associated with each object is specified at the time of registration. In the virtual light example, when the virtual light object turns on, all objects in the room which registered interest in it will have their own particular version of the method invoked to simulate the effect of a light source invoked automatically. Similarly, the designers of the MPCAL multi-user calendar system [Greif87] used *triggers* to specify additional computation to be performed when a particular computation is to be performed on a particular object. Implicit invocation is seen as a welcome addition to object-oriented languages in general [Notkin93].

Another form concerns the general definition of events. In the Interactive Multimedia Presentation platform [Bates94], multimedia events can be defined for multimedia objects by specifying a portion of the object. For example, a contiguous subset of video frames in a video data object could be defined as an event. When those frames are displayed, the display triggers a system-wide event. Event-driven rules can then be defined by associating actions with those events. Addition of this functionality was seen as necessary due to the event-driven nature of multimedia applications.

Event notification has also been integrated with the external storage of data in groupware applications. The DeVise multi-user hypermedia architecture [Gronbaek94] uses an object-oriented database to provide permanent storage for hypermedia objects and integrates its event notification with the functionality of the database. The type of events used here are related to locking and transactions on shared hypermedia objects.

Another fairly general form of notification is that of *news distribution*. This type of event notification is often referred to as publish-subscribe news dissemination. It is used in environments where the dissemination of fairly large-grained objects (news items) is important. Several systems cater solely to this type of event notification [TEK93].

2.4 Scalability

Object-oriented multi-media and virtual reality applications have the need for large numbers of fine-grained objects. For example, in certain multimedia frameworks, continuous-media data streams are modelled as collections of objects [Gibbs93]. In virtual reality applications, objects in a virtual world are often modelled as collections of finer-grained objects. For example, a building may be a composite of a roof object, wall, window and door objects, and contain furniture objects, etc.

Further, the introduction of a new participant into the groupware session may have the effect of creating new groups and adding members to existing groups. A multi-media conference may be attended by tens and possibly hundreds of participants. Thus, groupware applications need to scale in terms of the number of members allowed to participate in the groupware session.

High-latency networks, such as current wide-area networks, put different scalability requirements on groupware. Response times must be within the limits tolerable for real-time groupware applications.

There are other requirements which groupware applications in general require which we will not cover here: for example, persistence of shared data objects, and the need for roles in the provision of access control and security [Greif87]. For a wider discussion of the requirements of groupware applications, see [Greif87, Grudin89, Gronbaek93].

3 Object Groups - Design Issues

In this section, we consider design issues for object groups which are intended to satisfy the application requirements highlighted in Section 2. However, the issues really pertain to *object groups systems which implement an object group abstraction*. These include issues involving the definition and scope of the group abstraction itself, the design of the group membership and communication mechanisms, as well as properties required to support the implementation of higher level abstractions, such as replicated shared data, or event notification.

As we shall therefore be discussing design issues of object groups systems, we shall begin this section by classifying systems which support the object group abstraction based on the type of programming model they provide. We shall then investigate design issues of object groups systems and relate them back to the groupware application requirements we mentioned in Section 2.

3.1 Classifying Object Groups Systems

An *object group system* is software which directly supports the object group abstraction. It provides mechanisms for object groups to be created and for objects to join and leave the group. It also minimally provides a group invocation mechanism to allow efficient communication with group members.

With these minimal requirements specified, object group systems can provide varying degrees of support for programming with object groups. There are several object group systems in existence at the time of writing. Some commercially available, some only existing as research prototypes. They are largely differentiated by the programming model they provide, each with a different emphasis on how to use the object group abstraction to structure applications.

We have identified three different approaches to the implementation of the object group abstraction as follows:

Object Groups. The object group model provides the programmer with the abstraction of a group of objects, in the form of an *object group*, a single logical entity. The object group has mechanisms provided for allowing objects to join and leave the group and for allowing members of the group to communicate with each other. Object groups generally also provide an interface of operations which non-members of the group may invoke on the group as a whole, and these may vary in the degree of transparency they offer. Examples of object group systems include Reliable Distributed Objects [ISIS93, ISIS94] and Electra [Maffeis93].

Fragmented Objects. The *fragmented object* model aims to extend the object concept to a distributed environment. The abstract view of a fragmented object is of a single, shared object whose distribution is hidden to clients. The concrete view is of a collection of objects whose distribution and communication between 'fragments' (objects in the collection) is controlled entirely by the fragmented object designer. The fragmented object model encompasses several abstractions: grouping of fragments, specification of cooperation, specification of internal and external interfaces, and a mechanism for binding clients to the fragmented object. Examples of systems supporting fragmented objects are SOS [Shapiro89] and POF [Makpangou91].

Interface Groups. The ANSA [ANSA93] programming model provides the abstraction of a group of interfaces, or interface group. Interfaces figure highly in the ANSA model, being seen as the central means of providing information and access to service consumers(clients) in a distributed system. Interfaces are passed around the system by a distinguished application, the Trader. In this context, an interface group is a collection of interfaces which can be treated as though they were one, allowing clients to invoke operations on a collection of objects without knowing the exact membership of the collection or the location of the members. This capability provides the basis for the distribution of the implementation of a service over a set of interfaces.

Examples of each of these types of object group systems will be given in Section 4 - Case Studies.

3.2 Design Issues

There are two types of design issues which we need to consider when designing an object groups system which is to be suitable for structuring groupware applications. The first type includes issues which apply to the design of the object group abstraction itself. The second type includes issues related to the provision of event notification mechanisms, shared data and group-aware concurrency control mechanisms in terms of object groups which are also required. In order to emphasize the separation between an object group abstraction and the higher-level abstractions it intends to support, we treat these two groups of design issues separately.

3.2.1 Design Issues For Object Group Abstraction

Design issues which pertain to the group abstraction encompass *group membership* properties of object groups, such as the granularity of objects allowed, scalability, dynamic join and leave mechanisms, overlapping groups. Others pertain to the *group communication* properties of object groups, such as group invocation semantics and transparency of group invocation, and the ordering of invocations among group members.

Granularity of Objects. There is a requirement for support of both large-grained objects and fine-grained objects in object groups. Multimedia and virtual reality objects can be extremely fine-grained, whereas service components of the application, such as an object-oriented database management interface may need to be replicated to provide fault-tolerance of that component within the system.

Scalability. Object groups need to be able to scale to large numbers of objects per group and groups per application. Both virtual reality and multimedia applications have potentially thousands of objects interacting in an application. Scalability becomes a problem when dynamic joining and leaving of the group is a large factor in the application, and when there is a high incidence of overlapping groups, as described below. Further, the fact that groupware applications need to execute over wide-area networks will influence the design of object groups, particularly in the areas of naming and the design of remote object references.

Dynamic Join/Leave. There is a requirement for objects to be able to dynamically join and leave groups, especially when groups are used for membership purposes in applications. For example, if a light source moves through a virtual room, we may want to group together all objects which need to simulate a light shining on them for display purposes. The ISIS designers noted that high frequencies of joins and leaves in process groups can result in significant performance loss [vanRenesse93, Glade92].

Overlapping Groups. Objects very often need to be a member of several groups at a time. For example, an object may have registered interest in several events. In this case, the groups in question are said to *overlap.* When there are message ordering constraints on the groups in question, the ordering protocols guarantee that the messages will be ordered with respect to all overlapping groups. This common ordering may be required by the application for state consistency purposes. The ISIS designers have found that preserving ordering in the context of overlapping groups leads to severe performance degradation [vanRenesse93, Glade92], due to the increased delaying of messages to preserve the additional ordering requirements.

Open Groups. Open groups are required to allow modelling the provision of a *service* more closely, where those accessing the service (clients) do not have to be members of the group supporting the service. In the case of services with large numbers of clients, as with say a naming service replicated for fault-tolerance, it would be unreasonable for each client to have to join the group in order to access the service.

Group Invocation. The group invocation mechanism is perhaps the most important feature of object groups. It not only masks much of the complexity of communicating with a group of objects, but can be used to implement a wide range of object interactions, such as various types of event notification, replication of objects, and load balancing schemes, among others.

Group invocation can be seen as a combination of three separate activities: distribution of the invocation to one or more group members, processing of the invocation by those group members, and collation of one or more of the replies from each group member [ANSA93]. The choice and number of members involved in the invocation, the ordering imposed on the distribution of the invocation messages to those members and the way in which replies are collated determine the *group invocation semantics.*

The interface presented to non-group members by the object group need not be the same as the interfaces of the member objects. In fact, in a heterogeneous group, this will not be the case. Member objects should at least implement all of the methods in the interface presented to clients of the group.

There are two types of group communication: communication between group members themselves, and communication between non-group members and the group. In the case of peer groups, only group members can use the group communication mechanism. In some object group systems [Shapiro91, ANSA93b], group member objects are provided with two distinct object interfaces: a client interface which is used to accept invocations from non-group members, and a group interface use to accept invocations from group members.

The level of transparency in group invocation is an important issue: to what extent should the semantics of group invocation be configurable by the programmer. It is important that the programmer be able to specify different distribution and collection policies; for example, in an invocation to a collection of object replicas, we need only accept the first reply, as all others will be the same. On the other hand, the programmer shouldn't have to build the most common group invocation paradigms from scratch.

The invocation mechanism should also allow the programmer to specify the ordering semantics of the invocation, as this has an effect on the consistency of the application state. For in the distribution of invocations to group members, network communication delays or failures may result in invocation messages being delivered in an order other than that in which they were sent. This will be discussed in more detail in the section *Ordering Of Group Communication.*

Replica groups can lead to invocation semantics where a group of objects needs to make an invocation on a single object. Such object groups are referred to as *client groups* [ANSA93b]. For example, if an object containing a reference to another object is replicated, any invocation on the replica group which causes an invocation on the referenced object by one member will generate invocations by all other members. Generally, the invocation semantics required in such a case is for one invocation on the referenced object to be made, and the result distributed to the calling group members. The distribution and collation mechanisms of the invocation mechanism need to allow for such cases.

Two further anomalies arises in passing object references as arguments to invocations of replica groups [Achmat94]. In the context of remote invocation, an object reference is a local reference to remote objects which can be used to make invocations on that remote object. Many distributed object systems allow passing object references as arguments to invocations on remote objects [Shapiro91, ANSA93a, NeXT93]. Object references are often implemented using *proxies*[Shapiro86] which act as a local representative for the remote object.

If a client invokes a method on a replica group which takes a client-side object as an input argument, each replica gets an object reference to the client-side object. During processing of the method, each replica makes an invocation on its proxy for the client side object. This is another case of the client group invocation problem. Now, suppose a client invokes a method on a replica group which returns a server-side object as a return value. Each server will return an object reference for its return value. Because the servers are replicated, we would usually filter the replies from all servers and use only one reply value. In this case, that would mean keeping the proxy from one server only. When the client accesses the proxy, only one server side object gets updated. Both of these anomalies require special attention when designing the group invocation mechanism for replica groups.

Ordering Of Group Communication. The state of an object depends on its initial state, the invocations it receives after initialization, and the order in which those invocations are made. Because update operations generally are not commutative, changing the order of invocations can result in different, possibly unwanted, object state. In the case of an object group, if two clients make invocations on the group concurrently, it is possible that the invocations will be delivered to individual group members in different orders, possibly resulting in different object states. This is because invocations are delivered to remote objects using the network transport, which is subject to failures which can delay the delivery of invocations, causing invocations to be delivered in varying orders.

Object groups need to provide certain guarantees on the ordering of invocations delivered to group members to allow the application programmer to reason about the consistency of the application state. The exact requirements for maintaining consistency of application state are application-dependent. For example, in groups which depend only on correct group membership information, join and leave events need to be viewed across all members as consistently ordered with invocations.

The issue of message ordering and its use in distributed systems has been studied extensively in the literature [Lamport78, Birman89, Birman93a, Ladin90, Peterson89, Amir91]. The studies usually concern the properties of *multicast communication primitives* which provide one-to-many distribution of messages within the distributed system. There are generally three types of ordering of events which have been studied: FIFO, causal and total. In *FIFO*(first in, first out) ordering, invocations from a single client are guaranteed to be delivered at each group member in the order in which they were made. FIFO ordering makes no guarantees about the relative ordering of messages from different clients. In *causal* ordering, invocations are guaranteed to be delivered in an order which respects the causal ordering as defined in [Lamport78]. This ordering preserves causal relationships between invocations which arise from different clients. Causal ordering does not make any guarantees about the order of invocations by different clients which do not have a causal relationship. Finally, a total ordering guarantees that invocations by clients will be delivered in the same order at all object group members.

Most ordering protocols enforce ordering by delaying the delivery of messages which need to be ordered after messages which have not yet arrived. For this reason, a stronger ordering constraint usually brings slower effective communication performance. Thus, in many circumstances, it is desirable to choose the weakest ordering constraint possible which will preserve consistency of application state.

There has been a vigorous debate concerning ordering of events in distributed systems and the level at which ordering should be enforced [Cheriton93, Birman93b]. Many of the issues raised are relevant to the design of groupware applications using object groups.

For the purposes of object groups, it is important that several ordering alternatives be presented, so that the application designer may choose the most effective ordering for the application in question.

3.2.2 Design Issues For Higher Level Mechanisms

These are design issues which relate to the provision of mechanisms for event notification and replicated shared data. Neither is part of the definition of object groups, but we expect that the object groups system should allow easy construction of the various event notification mechanisms and provide replicated shared data and associated concurrency control mechanisms if they are to be useful in building groupware applications.

Event Notification Provision. Event notification can be seen as a specialised form of group invocation, with the object group being those objects which have registered interest in the event. Thus, the semantics of event notification can be fairly directly mapped onto an object groups programming model. The object groups system should provide a flexible group invocation mechanism, one which can be tailored to provide the several styles of event notification available.

Ordering of invocations is important in the case of event notification, as events in a groupware application may be causally related. On the other hand, they may not be. The group invocation scheme should therefore allow tailoring to handle these requirements. It may also be necessary to supplement the object groups system with registration mechanisms and mechanisms for defining events.

Shared Data Provision and Concurrency Control. As noted in the Section 2.2, shared data which is replicated at participant workstations is required for fast local access to shared data in a distributed environment.

The basic provision is that of an abstraction of a *replicated shared object*, with logical operations allowing the programmer to invoke the methods of the object. The implementation needs to ensure that non-update operations are satisfied locally, and update operations are transferred to other replicas in the group. When simulating a shared object using replicas which reside in distinct processes, a replicated object can receive concurrent invocations on distinct replicas. This leads to a requirement to manage both the *coherence* of the replicas and the need for group-aware concurrency control to ensure the *consistency* of the shared data.

Coherence of Replicated Objects

A design issue specifically related to the replicated nature of shared data concerns the *coherence* of the shared data replicas. A replicated data item is said to be *coherent* if the value returned by a read operation on the replicated data item is the same as the most recent write operation to the replicated data item. Coherence of replicated shared data ensures that all participants see the same shared values, even though those values may be implemented by distinct replicas on separate machines. This definition is usually applied to cases where data items are blocks of virtual memory [Li89] or file records on disk, where read and write operations are the primitive operations used to access the data. In the case of abstract data types such as objects, invocations may involve several primitive read and write operations, possibly on different objects where operations on composite objects are concerned. Extending the notion of coherence to this case, we say that a replicated object is coherent if the values of instance variables read by a non-update invocation are the same as the values written by the most recent update invocation.

The coherence of the replicated data needs to be maintained even in the presence of high-latency networks, as might be experienced in running a groupware application over a wide-area network. When an object invocation is made, the coherence mechanism needs to take account of any invocations currently in progress.

The maintenance of coherence of replicated shared data depends on the implementation strategy. Several methods have been developed:

Quorum-consensus. In the quorum-consensus methods [Thomas79, Gifford79], shared data replicas must vote on which operations are to be accepted for processing and a majority of votes is required for processing to take place. These methods were designed for managing replicated files in file servers and were integrated with transaction mechanisms provided by the file service. These methods have been adapted to manage replicated data based on abstract data types [Herlihy86].

State Machine. In the state-machine approach [Lamport78, Schneider90], replicated shared data is provided by replicating objects maintaining the data and coordinating client interactions with the server replicas. The approach relies on a protocol to ensure that all replicas receive all client invocations and in the same order. It has been shown that the ordering requirements can be relaxed in cases where the consistency of the application allows it [Birman89].

Shared Data-Object. In the shared data-object model [Bal88], shared data is provided in the form of data objects which can be shared between child processes of a parent process. Operations on data objects are atomic and implicit locking is used to guarantee serializability of concurrent operations on a single object. Transactions are not provided and are seen as a higher-level responsibility.

An object groups system needs to have the mechanisms available to provide an efficient implementation of the chosen method of replication. These include a flexible group invocation mechanism which provides several possible ordering guarantees as described earlier.

Consistency of Application State

Another issue which arises is *consistency* of the application state represented by the shared object. When subject to concurrent invocations without proper concurrency control, updates to the shared data can get lost or partially applied, resulting in an inconsistent state. Consistency of application state is ensured by controlling concurrent accesses to the shared application state in such a way that preserves consistency. The mechanisms needed to preserve consistency can depend on the presence of concurrent invocations on the shared object, and whether composite operations are required to take the object from one consistent state to another. However, the concurrency control mechanisms used to coordinate concurrent accesses to the shared data need to be *group-aware*, for reasons outlined in Section 2.2.

Ensuring consistency of shared data through appropriate concurrency control has been studied thoroughly in the case where the data is centrally located[Bacon92]. However, the distributed, replicated nature of the shared data and the requirement for concurrency control mechanisms to be group aware present significant design challenges.

The consistency problem can be looked at in two separate cases: the case where the shared data abstraction is subject to concurrent accesses by *single concurrent invocations*, and the case where the abstraction is subject to concurrent accesses by *composite concurrent invocations*.

Single Concurrent Invocations. By a single concurrent action, we mean a single invocation on the shared object which may take place concurrently with other single invocations. There are several possibilities for controlling concurrent access to the shared object which have been used in traditional concurrency control, namely locks, monitors and path expressions[Bacon92]. Of these, locking has seen perhaps the widest use in groupware applications.

In a scheme based on locking, exclusive access to a shared data item is gained by requesting and obtaining an associated lock on the object. Locking avoids the problems associated with concurrent access by forcing accesses to occur serially. It is generally associated with controlling access to data shared between processes on a single machine, such as memory-mapped shared data, or between threads within a single multi-threaded process. However, in using locking to achieve exclusive access to a replicated shared data object, the distributed nature of the replicated data and the requirement that the concurrency control mechanisms be group-aware complicate matters.

Locks have been used implicitly in the implementation of replicated shared data to provide a certain form of concurrency semantics, as for example in the shared data-object model [Bal88]. A non-deterministic serial order for concurrent invocations is established and invocations on the object are treated atomically. This serialization results in object semantics as when clients access a remote object in a client-server environment: whichever invocation arrived first is processed first. This effectively provides centralized object store semantics using the replicated object.

Explicit locks have been used more widely than implicit locks, and several group-aware versions have been developed. *tickle locks*, which are sensitive to idle processing time, and the relaxation of traditional consistency constraints have been investigated in the context of the Collaborative Editing System and the MPCAL multi-user calendar application [Greif87]. [Greenberg94] discusses *optimistic locks* which allow a lock to be obtained immediately without waiting for confirmation of granting, avoiding the interruption of work flow. Updates are made to the data item in question optimistically in the sense that updates will be discarded and the data item returned to its original state only if the lock is not eventually granted. *Visual cues* indicating when a shared data item is locked and by whom can also reduce contention for shared data items, by allowing users to negotiate, possibly through a computer-mediated interface [Ege87].

The lock abstraction has been extended to the distributed case[Bacon92]. However, many of the group-aware locking mechanisms developed have been used in centralized architectures and distributed counterparts of these locks have not been developed.

Composite Concurrent Invocations. By a composite concurrent action, we mean a collection of invocations on one or more shared objects which may take place concurrently with other invocations on those objects. For example, a composite action could be an invocation on a shared data object which contains shared data objects as sub-objects. Thus each may trigger off another invocation on a shared object of its own. Alternatively, the shared object(s) may be such that a collection of operations need to be performed atomically, or without interference from other actions in order to ensure consistency of the state represented by the object(s). A common example of where such a requirement arises is when a participant joins a groupware session, as described in Section 2.2.

In traditional concurrency control on shared data, transactions are used to provide recovery and serial equivalence [Bacon92], as well as atomicity of operations on shared data. Transactions allow the application programmer to define a collection of operations which take the state from one consistent state to another to be executed without interference. When distinct transactions execute concurrently and involve accesses to common shared data, the transaction mechanism executes the operations making up each transaction in such a way that the result is as if the transactions were executed in isolation, one after the other. This is the serializability property. Transactional serializability has been widely used in controlling concurrent access to shared data residing in a centralized repository where accesses are made through a single interface, such as a DBMS or file system. Transactions can be classified by the mechanism they use to decide upon when two concurrent transactions conflict: lock-based, optimistic, or time stamp-based.

Non-optimistic transactions, such as lock-based transactions and time stamp-based transactions, block the processing of concurrent transactions until a global serialization order of the transactions has been determined. Participants perceive this as a delay in the execution of the operation, which may be considerable. *Optimistic transactions* can be used to avoid delays in execution caused by enforcement of serial order of access to shared data items. A delay is incurred only if two transactions conflict and results from the need to either return to an earlier checkpointed state, or repair a state using undo operations [Greenberg94]. As in the case of optimistic locks, the advantage is the flow of action of the participant is not interrupted while waiting for the transaction to determine the serializability order. However, as [Greenberg94] notes, if the transaction eventually does need to be aborted, this can cause confusing and disrupting behaviour for the participant, as changes made will need to be undone.

In contrast to cases where the shared data is centrally located, a concurrency control mechanism based on transactional serializability for the replicated shared data abstraction would have to take into account the distributed, replicated nature of the shared data, as well as the need for group-awareness.

Work has been done on making transactions group-aware. The consistency requirements which transactions are intended to preserve can be relaxed when the participant is available to interact and make decisions concerning its execution. For example, some transaction-based systems allow data items which have been read during a transaction to be changed by other concurrently executing transactions. With *notify locks*, changes are not locked out but users are informed when data items in the read set of the transaction have been modified [Skarra86]. The user is informed and can then take corrective action, either by compensating for the changes or performing additional actions. Interactive transactions [Lee93] provide another approach where a transaction will apply itself iteratively on shared data until an application-dependent cooperative objective has been reached.

Several researchers have pointed out that the serializability property of transactions is not appropriate for groupware applications [Ellis87, Greenberg94]. Transactions have also been identified as unsuitable for directly distributed applications. [Birman91] has noted that this traditional form of concurrency control which works for centralized data fails for distributed data. The primary difficulty is that distributed data is not accessed through a common user interface, as in the case of a database or file system. Secondly, transactions do not take into account the semantics of object methods when managing concurrency. He advocates instead a model based on virtual synchrony and linearizability [Herlihy90].

As mentioned in Section 2.2, the type of concurrency control mechanism chosen has a considerable effect on the behaviour of the application as seen at user interface. We have seen that groupware needs flexible concurrency control primitives which minimise interruption to work flow (optimistic locks, optimistic transactions) and which are group-aware (interactive transactions, tickle locks). An object groups system which is to be used for structuring groupware applications will need to provide concurrency control mechanisms suitable for managing replicated shared data in a group-aware manner.

4 Object Groups - Case Studies

In this section, we shall examine several examples of object group systems which are commercially available or exist as research projects. We shall attempt to highlight the areas where the systems are strong for groupware support, and where they are lacking.

For each sample object group system, we shall highlight the following aspects:

* history of the project
* programming model/abstractions
* types of object groups supported
* support for replicated shared objects and/or event notification
* support for concurrency control

4.1 Orca

ORCA [Bal87] is a language for distributed programming which supports the *shared data-object* model: a programming model for distributed programming which allows parallel applications on loosely coupled distributed systems to share data. ORCA is not an object group system *per se*; however, we include it here as it supports replica object groups implicitly.

ORCA forms part of the work undertaken by the Amoeba project at the Vrije Universitaet in the Netherlands during the mid -1980's. The ORCA model was intended for medium-grained to large-grained parallel applications. It does not intend to support fine-grained objects.

Programming Model/ Abstractions

On a multiprocessor system, data is shared among processes running on separate processors through physical memory shared between the processors. In a loosely coupled system, processors do not share memory and must generally communicate via some form of IPC. ORCA provides logically shared data-objects by managing a physical implementation based on replicated data.

ORCA is an implementation of the *shared data-object* model. According to the shared data-object abstraction, data is encapsulated in data-objects, which are instances of user-defined abstract types. Each data object instance provides a set of operations to allow manipulation of the data object state.

The shared data-object model requires:

(i) all operations on a given object are executed atomically ;
(ii) all operations apply only to single objects, so an operation invocation can modify at most one object

Processes communicate through shared data objects. Objects declared local to a process may be shared between child processes by passing them as shared parameters when the child process is created. Child processes are created in ORCA using the fork command, which allows the resulting child process to be run on a selected CPU.

Types Of Object Groups Supported

ORCA supports replica groups only. These replica groups are implicitly defined and managed by the ORCA run-time system. They are active replica groups - updates are applied to all copies.

Support For Replicated Shared Objects

ORCA provides support for replicated shared objects, but with limitations due to the application domain it addresses. Medium and large grained objects are supported, but fine-grained objects less so. Access to replicated objects is limited to child processes accessing a shared data-object of a common parent. This does not constrain sharing to a single machine, however, as the Amoeba distributed operating system [Mullender90] is based on the processor pool model: child processes may execute on different CPUs in the pool. Finally, operations on data-objects are constrained to affect only the data-object in question - composite shared data objects are not supported.

Support For Event Notification

Event notification is not supported between data objects. However, in order to allow cooperation between processes sharing data objects, ORCA provides a condition synchronization mechanism which allows processes to block until a condition becomes true. This mechanism is integrated with the operation invocation model by allowing operations to block on conditions. A blocking operation consists of one or more guarded commands.

Support For Concurrency For Shared Objects

Mutual exclusion of concurrent operations on shared data is ensured by arranging that operations on shared objects execute indivisibly: if two processes concurrently invoke an operation on an object, one is selected non-deterministically and executed in such a way that serializability is guaranteed. Even though individual operations are indivisible, sequences of such operations are not; such a mechanism for providing indivisibility of sequences of operations is seen as the responsibility of the programmer.

This is the model of concurrent access described in Section 3.2

4.2 Fragmented Objects

The Fragmented Object (FO) model is an extension of the object concept to a distributed environment [Shapiro91, Makpangou93]. The abstract view of a fragmented object is that of a single, shared object. The concrete view is that of a collection of objects whose distribution and communication between other objects is specified by the fragmented object designer.

The initial work on fragmented objects was done in the context of the SOR project at INRIA in France. The goal of the project was to examine object-support operating systems: an operating system which would provide low level support for remote object invocation, migration of objects, and application support for object-oriented application design.

Programming Model / Abstractions

From the client's point of view, a fragmented object is a single, shared object. The distributed nature of a fragmented object is hidden to clients.

From the designer's point of view, a fragmented object consists of the following elements:

Fragments. A set of elementary objects making up the fragmented object;

Client Interface. An invocation interface which client objects use to make invocations on the fragmented object. The client interface may vary from client to client;

Group Interface. An invocation interface which fragments make available to each other;

Connective Objects. Low-level, shared fragmented objects used for communication and cooperation between fragments

The fragments may be spread over several address spaces. Fragments cooperate to maintain a consistent view of the single logical entity that the client sees. For example, if the fragmented object represents a replicated shared object, it is up to the fragments to maintain coherence and consistency of the shared data among themselves.

A client of the fragmented object needs to obtain a proxy [Shapiro86] from the fragmented object in order to make invocations. The proxy returned may be different for each client, depending on its access needs and rights. A proxy provides a network transparent invocation interface to the fragmented object which is local to the client's address space.

To allow a client to obtain a proxy object for a fragmented object, there are three styles of binding available: local binding for statically configured services, a system-defined binder (similar to the ANSA trader) for dynamic location of target objects, and a dynamic binding style based on a per-FO *provider* object. In the dynamic case, a binding takes place in three steps: first, a name lookup yields a provider object for the named interface; second, a binding request is forwarded to a particular method of the provider object; and third, the provider object dynamically instantiates a proxy implementation in the client address space. This proxy may be chosen based on the client's identity, binding request arguments, the load of the client's host, and other criteria.

Connective objects are used by the fragmented object designer to allow the internal fragments to communicate and cooperate to provide the service required of the fragmented object. There are three types of connective objects: *channel*, *sharing* and *synchronization.*

A channel connective object is a lower-level FO which allows fragments to communicate according to the client-server model. A (client) fragment may use a local channel interface to make invocations on the group interface of a (server) fragment. There are point-to-point channels for communication between pairs of fragments, and multi-point channels for one-to-many communication between fragments. Point-to-point channels include rpc-style (synchronous) as well as asynchronous communication. Multicast channels include FIFO, causal and total ordered versions of reliable multicast communication.

A sharing connective object implements a specific sharing abstraction, such as shared memory, replicated objects, partitioned data structures and streams. It provides fragments with a memory-like interface which is used by some to write (producers) and by others to read what was written (consumers). The sharing connective objects provide type-checked access to the memory in question and also integrated concurrency control, selected from a number of possible concurrency control policies.

Synchronization connective objects provide implementations of several synchronization primitives, such as distributed token passing, semaphores, barriers, and monitors.

The BOAR library [Makpangou91] is a collection of elementary connective object types to aid the programmer in structuring applications built using FOs. The BOAR library includes connective objects described above.

Types Of Object Groups Supported

The FO model allows the fragmented object designer to create object groups of a wide range of types. This is due to a versatile implementation of the fragmented object abstraction, allowing the programmer control over the important aspects of a distributed object: the interfaces it presents, the placement of its fragments, communication between fragments, and how clients bind to the object.

Fragmented objects may contain fragments instantiated from arbitrary (C++) classes, allowing homogeneous or heterogeneous object groups. Both open and closed groups are possible: open groups are those which include client proxies, closed groups those which do not. However, though closed groups could be constructed, it seems that they go against the idea of a fragmented object. In that case, a fragmented object is somewhat like an active object with no client interface.

Fragments join and leave fragmented objects dynamically, but only through instantiation as part of a fragmented object class definition. Thus, the types of object instances which may join a fragmented object are determined by class, rather than by the interface they support. This is true to the idea of a fragmented object as a well-defined functional entity. However, it puts very strict requirements on who is allowed to join a group dynamically: the groups are heterogeneous in a restricted sense. It also makes the support of membership groups difficult. Restricting membership to objects which conformed to a particular set of protocols would give added flexibility but detract from the idea of an object specified by a class definition.

Client proxies provide network-transparent access to the fragmented object and hide the inner workings of the FO. Thus transparent groups are directly supported. Non-transparent groups could be supported by allowing the client proxy to return group membership information.

Overlapping groups seem not to be supported in this model: that is, an object cannot be a fragment of two fragmented objects at the same time. This is primarily due to the nature of the class definition of the object group as described above.

Support For Replicated Shared Objects

The fragmented model supports replicated objects directly through sharing connective objects.

For example, there is direct support for replication of the following abstract data types: simple data types, records, untyped raw memory access, lists, trees and graphs. Replication consistency policies and synchronization policies may be chosen to suit the application.

Further, quite general replica groups can be constructed by the fragmented object programmer using channel and synchronization connective objects. A sample implementation of a replicated file service is described in [Makpangou93].

Support For Event Notification

Event notification is not directly supported, in that there are no pre-defined FOs in the BOAR library to provide the type of event notification functionality as described in Section 2. However, the extremely versatile implementation of the fragmented object abstraction would make writing a collection of event notification classes relatively simple. However, the fact that overlapping groups are not supported will very likely constrain the implementation.

Support For Concurrency Control For Shared Objects

As mentioned earlier, certain support for concurrency control exists in the form of pre-defined (distributed) synchronization connective objects. These can be used together with sharing connective objects to provide different concurrency control policies for the replicated data.

4.3 ANSA (Advanced Network Systems Architecture)

The ANSA Model For Interface Groups [ANSA93b] is an abstraction which aims to provide an easy-to-understand programming structure for replicated services, as well as resolving certain distribution issues.

The ANSA project [ANSA93a] dates from 1988 in Cambridge, England. It consists of a consortium of companies with the aim of proposing an architecture for networked computer systems to support distributed applications and to promote the acceptance of the results as an industry-wide standard. ANSA is an architecture or framework within which the design and implementation of distributed computer systems can take place. It has the specific objective of enabling application components to work together, despite a diversity of programming languages, networks, computer hardware, communication protocols, security policies and management policies.

ANSA is not designed to support object-oriented distributed applications, even though the concept of an object exists in the framework. Instead, ANSA focuses on interfaces and attempts to define all characteristics of an application statically through the use of an IDL to define the interface and its properties. The model of an object in ANSA is one in which an object has interfaces rather than operations. The various interfaces are views of the potential operations of an object. An object also provides encapsulation of its internal state and it is possible to define an interface that acts both as an abstract type and also a generator for a class of interfaces, much like classes in class-based object-oriented languages.

Programming Model / Abstraction

The interface group abstraction is to treat a number of interfaces as though they were one, allowing a client to invoke operations on a collection of objects without being aware of their number or location. An *interface group* is a collection of one or more interfaces which are accessible externally through a single service interface. All the interfaces in an interface group must conform to a common interface type, which will be used as the type of the interface group service interface. In addition, a management interface is associated with the interface group to allow group management activities to take place, such as requests for joining and leaving the group.

In the interface group model, a great emphasis is placed on the implementation of transparent invocation through an orchestrated set of mechanisms. Mechanisms provide services through interfaces and are partially defined by policy objects. Choosing different policy objects for a single mechanism defines different behaviours for that mechanism. A number of mechanisms are required to support transparent client-group interaction: *distribution* is the process of sending an invocation to all members of an interface group; *collation* is the process of reducing multiple messages from a group into a single message; *sequencing* involves group members deciding on the order in which invocations will be delivered; *quorum determination* involves ensuring consistency and fault-tolerance requirements by determining that the correct number of interface group members have received an invocation. Other mechanisms exist to control consistency of interface groups, to handle checkpoints of member state, and the introduction of new members to the group.

When an interface group is created, the functionality of the group is determined by specification of a particular collection of mechanism and policy interfaces. In particular, group templates can be used to specify the behaviour of an interface group. A group template consists of a service interface, a management interface, and a collection of mechanism and policy tuples. The behaviour of an interface group will be determined by its service type and the sequence of mechanism-policy pairs. This leads to a wide selection of group functionality

Binding to an interface group is performed by the client specifying a set of mechanism-policy pairs, which can be used by the binder mechanism to verify that the mechanisms and policies required by the client are consistent with those specified in the group template of the interface group being bound to. A group reference is returned which the client application may use to make invocations on the group service interface. The group reference however becomes stale when group membership changes occur. An invocation made with a stale group reference will result in an exception and the client application is forced to refresh the interface reference through a call to the ANSA relocation service. This creates problems for interface groups with a high number of joins and leaves.

Concurrency control in ANSA is generally achieved through the use of path expressions, through which server interfaces can specify concurrency and ordering relationships between collections of service operations. For interface groups, a concurrency manager mechanism is available to coordinate and enforce consistent behaviour between member interfaces, with different concurrency policies that can be plugged in.

Types Of Object Groups Supported

The interface groups model supports transparent active replica groups directly. This reflects a primary aim of interface groups as providing support for replication of ANSA services.

Due to the detailed consideration of the mechanisms which underlie the implementation of a group abstraction, new types of groups may be constructed by selecting appropriate policies and mechanisms which provide more direct access to the underlying communications as described above.

An object in ANSA may export several interfaces. It is possible for a single object to represent two members of a single interface group. As well, one object may be a member of several interface groups, thus allowing overlapping groups. ANSA does provide consistency management mechanisms for managing consistency of members within a single group, but it is unclear whether consistency can be managed across interfaces groups, as is required with groups which contain common members.

Support For Replicated Shared Objects

As mentioned earlier, there is direct support for replicated shared objects. The replication may be achieved through passive replica groups or active replica groups. However, passive replica groups are not of interest in the case where replicas are intended to be local to clients in order to provide fast local non-update invocations.

Replicas can be collocated with the application components they support. A replica joins an interface group by acquiring an external manager interface from the name service for the group and invoking an addMember operation, specifying an interface to be added to the group. After the joining interface is type-checked, the new member receives a copy of the group management and service references. It is unclear as to whether these references also become stale when new members join and leave the interface group.

ANSA directly supports the notion of a *client group*, which arises when an invocation on a replica group requires that all replicas within the group make a further invocation on a single external object or even another interface group. The need for client groups arises when any member of a replica interface group contains references to external objects.

Support For Event Notification

Interface groups do not provide direct support for event notification; however, it is possible to build event notification mechanisms using existing mechanisms. Subscribers need to provide an interface to join the group in order to receive notifications (or possibly implicit invocations).

However, due to the way in which group references work, each time the membership of the group of subscribers changed, the publisher would have to go through the relocation service to obtain a new group reference. This severely limits the suitability of such a mechanism in the case where the collections of subscribers is constantly changing.

Support For Concurrency Control For Shared Objects

As mentioned earlier, support for concurrency within group members is provided with path expressions, and consistency of the state of the interface group as a whole is managed by a concurrency manager interface.

It is not clear how concurrent invocations by clients on shared interface groups are handled, although the concurrency control manager could adopt a serialization policy for such concurrent invocations.

The designers note that any ordering which is used by the intra-group communication protocol needs to be made available to the concurrency control logic so that any intended determinism (order) may be maintained in the presence of concurrent execution of operations on member interfaces. Currently, multi-threaded active replica group members are not supported.

4.4 Electra

Electra [Maffeis93] is a toolkit for object-oriented, distributed programming in C++ based on the abstractions of *object groups*, *remote method calls* and *smart proxies*. It aims to provide high-level support for the object groups abstraction in a highly portable fashion.

The Electra toolkit is being developed at the Department of Computer Science, University of Zurich for the past several years. Various publications have been presented on the Electra toolkit [Maffeis93, Maffeis94] detailing its design approach. Electra is an ambitious project and is in its early stages of development at present.

One of the goals of Electra is to implement object groups using functionality from existing distributed control technologies (DCTs), such as ISIS, Horus, Amoeba, and Transis. The idea is to reuse the sophisticated group multicast and group membership functionality of these systems in such a way that applications written on one platform can easily be ported to another. A further goal is to be able to integrate new DCTs into Electra as and when they become available.

In order to achieve this goal, the abstractions making up the programming model and the programming tools described are built on top of a *virtual machine* (VM) layer. The VM layer is in the form of an abstract C++ class and contains primitives for handling threads, message passing, reliable multicast, thread synchronization, and group management. The VM layer separates and insulates the implementation of the high-level abstractions from differences between DCTs.

An *adaptor object* contains the implementation mapping the VM primitives onto a particular distributed control technology platform (ISIS, Horus, Amoeba, Transis, etc.). Adaptor objects are implemented as subclasses of the abstract C++ class specifying the VM. For each DCT, there will be an adaptor object which maps the VM primitives down onto the primitives of the particular DCT platform. DCTs differ in the primitives that they provide, and some adaptor objects may have to implement sophisticated functionality which exists in the VM layer but that the DCT does not provide.

The Electra toolkit consist of a run-time environment , a CORBA-style service definition language (SDL), a collection of class libraries, and a set of programming tools to monitor and control distributed applications. Object interfaces are specified in the SDL which allows inheritance of interface specification.

Programming Model / Abstractions

The Electra programming model is based on *object groups, remote method calls* and *smart proxies*. Objects in Electra can be passive or active (multi-threaded).

Remote method calls are implemented by *proxies* [Shapiro86], or local stub objects representing a remote service. Electra supports three types of remote method call: asynchronous, synchronous and promises. Promises are effectively asynchronous invocations which allow the programmer to explicitly synchronize with the returned result (if any) using a join-type primitive.

Smart proxies perform the basic invocation functions of ordinary proxies but can also provide certain transparencies such as caching of invocation results, masking the replication of remote objects, dynamic load balancing, and other functions.

Electra objects and object groups are identified by names of the form <domain>:<path>, where domain is a standard Internet domain name, and path is a regular UNIX path name. The name of a remote object (or object group) is required to instantiate a proxy for that object. A 'master-trader' service exists which manages the identification names for a domain. An identification name is required to instantiate a proxy for a service.

The Electra SDL is used to define CORBA-style object interfaces to specify remote method call semantics. At present, support is provided for specifying group communication ordering semantics, as well as collation of results from invocations which return multiple results.

Types Of Object Groups Supported

Electra supports transparent replica groups at present. Details of support of other types of object groups is not available but, as Electra is a programming toolkit, it is likely that explicit groups and other types of groups will also be supported in future.

Support For Replicated Shared Objects

The Electra SDL supports active replicated objects directly. The application programmer defines a non-replicated implementation of the object. In addition, an SDL definition for the object's interface is provided to specify the *category* of each of the methods of the interface. The category determines how member objects will receive invocations on the object group. For example, if the category of an invocation is cbcast, then an invocation of that operation will be multicast to all members using causal ordering. If the category of an invocation is method, then such an invocation will only be sent to one member of the group. Other supported multicast categories include abcast for totally ordered multicast, and ubcast for unordered multicast.

Replicated services are remotely instantiated by an Electra service called Mushroomer, which runs on each Electra host.

Support For Event Notification

Electra does not appear to provide direct support for event notification at present.

Support For Concurrency Control For Shared Objects

Although Electra supports active objects, little is known about the concurrency control schemes which are planned. As it is based on DCTs which provide loosely synchronous and virtually synchronous executions, it seems likely that any concurrency control will be built on these mechanisms, as will be described in Section 4.5.

However, linearizability is mentioned as a favourable approach to providing a correctness condition for concurrent objects.

4.5 Reliable Distributed Objects

Reliable Distributed Objects (RDO) is a collection of classes and libraries which provide an object-oriented interface to much of the functionality of the ISIS Distributed Toolkit. There are two versions of the system, one for the C++ language and one for Smalltalk-80. RDO is a full-featured object groups implementation.

RDO [ISIS93, ISIS94] was released in 1993 by Stratus Computer Inc., who purchased ISIS Distributed Systems, Inc. in December 1993. The ISIS Distributed Toolkit, to which RDO acts as an object-oriented front end, was the first family of reliable distributed computing products, and one of the main proponents of the process groups application architecture. RDO was preceded by similar work on object groups [Birman91].

RDO is implemented on top of the ISIS Distributed Toolkit. Programs which use the ISIS RDO/C++ *are* ISIS programs, with some C++ support for remote method calls and object groups.

Programming Model / Abstractions

The main abstractions are the object and the object group, both in a distributed context.

One of the central provisions of the package is a mechanism providing Remote Method Call (RMC) functionality to facilitate the writing of distributed object-oriented programs. Server objects define an interface and publish that interface to allow client objects to make remote method calls. The implementation of RMC is based on the proxy/dispatcher implementation. An Interface Definition Language (IDL) is used to define server interfaces and generate the client proxy stubs and server dispatcher stubs.

One interesting aspect of the remote method call implementation is that it is fully group-based: to make an object available for remote invocations, it becomes the member of a named singleton group. A client object wishing to make invocations on the remote object uses the group name to instantiate a proxy for it. Every remote object is represented as an object group, which means that there is a uniform way to communicate with remote objects. This avoids the problems of using a mix of point-to-point and group communication.

RDO also includes classes to easily support fault-tolerance models such as replicated objects and coordinator-cohort, the master-slave distributed execution model, and the publish-subscribe model of event notification.

The implementation of RMC is based on the proxy/dispatcher implementation. The package uses an Interface Definition Language(IDL) and associated IDL compiler to generate the client-side proxy and the server-side dispatcher classes, as well as marshalling functions for any abstract data types defined in the interface. The RDO IDL uses CORBA IDL-style syntax rather than C++ syntax to describe class interfaces.

Types Of Object Groups Supported

Support is provided for transparently replicating server objects to form replicated server groups, as will be described below. Additionally, there are classes which allow access to alternative design architectures which are supported by ISIS. For example, the coordinator-cohort classes allow services to be handled by a single object, the coordinator, but in the event that the coordinator object fails, one of a collection of cohort objects will take over and replace the coordinator.

RDO also allows the programmer direct access to group structures and mechanisms, as in the ISIS style, which allows the explicit group style of programming, used for writing directly distributed applications.

Support For Replicated Shared Objects

In order to support the almost transparent nature of the object replication mechanism, the IDL generates two implementations for every method defined in an interface: a single-reply version of the method, and a multiple-replies version. The multiple replies version of the method is used when communicating with a replicated object. For every return argument of the method defined in the interface, the multiple replies version returns a sequence of the appropriate data type, allowing the client to collect and inspect all replies from all replicas.

The group-based addressing mentioned earlier makes it very easy to introduce replication of a server object: in fact, instantiation of an object replica is achieved using the same procedure as instantiation of the original remote object. If a first instance already exists, the second instance just joins the same named group. Combined with the fact that a client side proxy will communicate with the group containing the object, this ensures that all replicas get the same message sends.

Support For Event Notification

RDO supports two event notification mechanisms, monitors and watches. An object can associate an event with an action routine (object invocation) and an associated parameter. When the event occurs, the invocation is made, passing the supplied parameter as an argument. In the case of a watch, the action routine is triggered only on the first occurrence of the event, whereas a monitor triggers each time the event happens, until the monitor itself is cancelled.

Also, RDO implements a comprehensive publish-subscribe news group system, called RDO News, which is suitable for news dissemination applications. In the News programming paradigm, news is organized in a way similar to USENET Newsgroups: news, or information, is organized by subject, and clients may holds any number of news subscriptions, and may also post messages to a news group.

Support For Concurrency Control For Shared Objects

Concurrency control in RDO is at present based on the virtual synchrony [Birman87] approach used in the ISIS Distributed Toolkit [ISIS92]. This approach uses the model of a loosely synchronous execution of the system which has been relaxed slightly in the case of concurrent events to allow the programmer to model processing events occurring in the distributed system. The guarantees that virtual synchrony provides concerning the ordering of invocations allows the programmer to keep shared state synchronized, at the granularity of individual invocations on shared data. However, the notion of a collection of invocations which are executed atomically on shared state is absent from this model.

5 Summary

In this paper, we have examined the requirements presented to the designer of real-time, groupware applications and have used those to formulate a set of design issues for object groups. Although centralized architectures have been used for groupware applications, the discussion was directed at groupware using a directly distributed, replicated architecture.

The primary requirements presented by real-time, groupware applications are the need for a group abstraction, the need for provision of replicated shared data and concurrency control mechanisms for managing concurrent access to that shared data which are group-aware, and the need for the application architecture to scale to large numbers of users and wide-area networks.

The design issues identified for object groups reflected these groupware requirements. The group abstraction should provide efficient implementations of group membership and group communication protocols to allow for high frequency dynamic joins and leaves of objects to and from the group. Objects also need to be able to be members of several groups at a time, and so the ability for groups to overlap is a requirement which has important consequences for performance. The abstraction should also be scalable in terms of the number of members within each group, the number of groups allowed, the number of groups that overlap.

In order to provide a flexible event notification functionality which can be tailored to the many types of notification required, the group invocation mechanism should be sufficiently flexible to allow ordering of invocations between members to be specified.

A core requirement is the provision of shared data and the concurrency control mechanisms made available to the groupware application programmer. There are several design options available for the provision of shared data, both for maintaining the coherence of shared data replicas, and managing the consistency of the shared state represented by the shared data. Group-aware concurrency control mechanisms are required which take into account the distributed and replicated nature of the shared data. Several mechanisms have been used in the past for controlling access, including group-aware locks and group-aware transactions. However, the serialization property is considered inappropriate for groupware, and other mechanisms which take into account the distributed nature of the shared data, and the semantics of the object invocations are being considered.

Five existing implementations of object groups systems were examined in Section 4. We found on the whole that support for the requirements outlined for object groups is weak. This is, however, not surprising as object groups have only recently been introduced as a mechanism for structuring distributed applications. All systems provided some support for replicated shared data, but were weak in the provision of concurrency control. None of the systems examined provided group-aware concurrency control support.

We have by no means covered all requirements for groupware applications is a distributed setting, but tried to highlight those which had the greatest bearing on the design of object groups. For example, [Grief87] has identified the need for roles in groupware with access control and security. In general, security will need to be implemented at some level. Work is being done on the requirements for secure process groups [Reiter92]. Thus, a security model could possibly be integrated as part of an effective object groups implementation.

Providing the right form of object groups abstraction suitable for groupware applications, together with the necessary higher-level abstractions such as event notification, replicated shared data and appropriate group-aware concurrency control mechanisms will make object groups an attractive mechanism for structuring distributed, real-time groupware applications. This work has attempted to highlight the design issues that need to be tackled in order to achieve that aim.

Acknowledgements

The author would like to thank George Coulouris, Jean Dollimore and Tim Kindberg who read earlier drafts of this report and whose comments were of great value in deciding its presentation. Any errors or omissions contained in the report are the sole responsibility of the author.

Bibliography

[Achmat94] Object Groups For Groupware Applications,
R. Achmatowicz,
Computer Science Department,
Queen Mary and Westfield College, University of London,
July 1994

[Amir91] Transis: A Communication Subsystem For High Availability,
Y. Amir, D. Dolev, S. Kramer, D. Malki,
Technical Report TR 91-13,
Computer Science Department, Hebrew University Of Jerusalem,
November 1991.

[ANSA93a] An Overview of ANSA,
R.J. van der Linden,
Architecture Report 000.00,
Architecture Projects Management Ltd,
May 1993

[ANSA93b] A Model for Interface Groups,
E. Oskiewicz, N. Edwards,
Architecture Report 002.01,
Architecture Projects Management Ltd,
February 1993

[Bacon92] Concurrent Systems: An Integrated Approach to Operating Systems, Database and Distributed Systems,
J. Bacon,
Addison-Wesley, 1992

[Bal87] Orca: A Language for Distributed Programming,
H. Bal, M.F. Kaashoek, A.S. Tannenbaum,
Report IR-140,
Dept. of Mathematics and Computer Science, Vrije Universiteit,
Amsterdam, The Netherlands
1987

[Bal88] Distributed Programming with Shared Data,
H. Bal, A.S. Tannenbaum,
IEEE Conference on Computer Languages,
IEEE, 1988

[Bates94] A Development Platform for Multimedia Applications in a Distributed,
ATM Network Environment,
J. Bates, J. Bacon,
Technical Report, University of Cambridge Computer Lab.,
Cambridge, England, 1994

[Birman87] Exploiting Virtual Synchrony In Distributed Systems,
K. Birman, T. Joseph,
11th ACM Symposium on Operating Systems Principles,
December 1987

[Birman89] Exploiting Replication In Distributed Systems,
Distributed Systems, 1st Edition,
Sape Mullender, ed.
Addison-Wesley, 1989

[Birman91] Maintaining Consistency In Distributed Systems,
Cornell University Technical Report TR91-1240,
November 1991

[Birman93a] The Process Group Approach To Reliable Distributed Computing,
K. Birman,
Communications Of The ACM, Vol. 36, No. 12,
December 1993

[Birman93b] A Response To Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication,
K. Birman,
ACM Operating Systems Review,
October 1993

[Cheriton85] Distributed Process Groups In The V Kernel,
D. Cheriton, W. Zwaenepoel,
ACM Transactions on Computer Systems Vol. 3, No. 2
May 1985

[Cheriton93] Understanding The Limitations of Causally and Totally Ordered Communication,
D. Cheriton, D. Skeen,
Proceedings of 14th ACM SOPS,
August 1993

[Cooper85] Replicated Distributed Programs,
E. C. Cooper,
Proceedings 10th Symposium on Operating System Principles,
Nov. 1985.

[CORBA93] The Common Object Request Broker: Architecture and Specification,
Dec. 1991. Revision 1.1, OMG Document Number 91.12.1

[Coulouris94] Distributed Systems: Concepts and Design,
G. Coulouris, J. Dollimore, T. Kindberg,
Addison-Wesley, 1994

[DIVE93] DIVE - The Distributed Interactive Virtual Environment Technical Reference Manual,
M. Andersson, C. Carlsson, O. Hagsand, O. Stahl,
Sweedish Institute of Computer Science,
Kista, Sweeden
July 1993

[DIVISION93] dVS Technical Overview,
Division Limited,
Bristol, England
1993

[Ege87] Design and Implementation of GORDION, An Object-based Management System,
A. Ege, C.A. Ellis,
Proceedings of 3rd Int'l Conference on Data Engineering,
IEEE, New York, 1987.

[Ellis89] Concurrency Control In Groupware Systems,
C.A. Ellis, S.J. Gibbs,
Proceedings of ACM SIGMOD, 1989

[Ellis91] Groupware: Some Issues and Experiences,
C.A. Ellis, S.J. Gibbs, G.L.Rein,
Communications of the ACM,
January 1991

[Fahlen93] Virtual Reality and the MultiG Project,
L. Fahlen,
Virtual Reality 93: Proceedings of Third Annual Conference on Virtual Reality,
London,
April 1993.

[Gibbs93] Visual Objects, chapter on Data Modelling Of Time-based Media,
S. Gibbs et al.,
University Of Geneva, 1993.

[Gifford79] Weighted Voting For Replicated Data,
D. K. Gifford,
Proceedings 7th Symposium on Operating Systems Principles,
ACM, December 1979.

[Glade92] Light-weight Process Groups in the ISIS System,
B. Glade, K. Birman, R. Cooper, R. van Renesse,
Proceedings of the Open Forum '92 Technical Conference,
Utrecht, The Netherlands,
November 1992

[Greenberg91] GroupSketch: A Multi-user Sketchpad for Geographically Distributed Small Groups,
S. Greenberg, R. Bohnet,
Proceedings of Graphics Interface '91,
Calgary, Alberta, June 91

[Greenberg92] Issues and Experiences Designing and Implementing Two Group Drawing Tools,
S. Greenberg, M. Roseman, D. Webster, R. Bohnet
Proceedings of Hawaii International Conference on System Sciences,
Kuwaii, Hawaii,
January 1992, IEEE Press.

[Greenberg94] Real-time Groupware as a Distributed System: Concurrency Control and
Its Effect on the Interface,
S. Greenberg, D. Marwood,
Technical Report, University of Calgary,
Calgary, Alberta,
February 1994

[Greif87] Data Sharing In Group Work,
I. Greif, S. Sarin,
ACM Transactions on Office Information Systems, Vol. 5, No. 2,
April 1987

[Gronbaek93] CSCW Challenges: Cooperative Design in Engineering Projects,
K. Gronbaek, M. Kyng, P. Morgensen,
Communications Of The ACM, Vol. 36, No. 6,
June 1993

[Gronbaek94] Hypertext Systems: A Dexter-based Architecture,
K. Gronbaek, J.A. Hem, O. L. Madsen, L. Sloth,
Communications Of The ACM, Vol. 37, No. 2,
February 1994

[Grudin89]	Why Groupware Applications Fail: Problems In Design and Evaluation, J. Grudin, Office: Technology and People, Vol 4, No. 3, 1989
[Hagsand91]	Object Groups: An Approach To Reliable Distributed Systems, O. Hagsand, H. Herzog. K. Birman, R. Cooper Technical Report, Cornell University Ithaca, New York, 1991
[Herlihy86]	A Quorum-Consensus Replication Method For Abstract Data Types, ACM Transactions on Computer Systems, Vol. 4, No. 1, February 1986
[Herlihy90]	Linearizability: A Correctness Condition For Concurrent Objects, ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, July 1990
[ISIS92]	The ISIS Distributed Toolkit Reference Manual, ISIS Distributed Systems, Inc.
[ISIS93]	Isis Reliable Distributed Objects for SmallTalk-80, RDO/ST Reference Manual, Aug. 93 Isis Distributed Systems, Inc.
[ISIS94]	Isis Reliable Distributed Objects for C++, RDO/C++ Reference Manual, Feb. 94 Isis Distributed Systems, Inc.
[Kaashoek91]	Group Communication In The Amoeba Distributed Operating System, M.F. Kaashoek, A.S. Tannebaum, IEEE Proceedings 11th Conference on Distributed Computing Systems, IEEE 1991
[Knister90]	DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors, M. Knister, A. Prakash, CSCW '90 Proceedings, October 1990.
[Ladin90]	Lazy Replication: Exploiting The Semantics of Distributed Services, R. Ladin, B. Liskov, L. Shrira, Proceedings of 10th ACM Symposium on Principles of Distributed Computing, August 1990
[Lamport78]	Time, Clocks and the Ordering of Events in a Distributed System, Communications of the ACM, Vol. 21, No. 7, July 1978
[Lauwers90]	Collaboration Awareness In Support of Collaboration Transparency: The Next Generation of Shared Window Systems, J.C. Lauwers, K.A. Lantz, ACM Proceedings of the SIGCHI Conference on Human Factors In Computing, Seattle, Washington, April 1990.
[Lee93]	A Framework for Controlling Cooperative Agents, K.-C. Lee, W. Mansfield Jr., A. Sheth, IEEE Computer, July 1993

[Li89]	Memory Coherence in Shared Virtual Memory Systems, K. Li, P. Hudak, ACM Transactions on Computer Systems, Vol. 7, No. 4, November 1989
[Maffeis93]	Distributed Programming Using Object Groups, S. Maffeis, IFI TR 93.38, Dept. Of Computer Science, University Of Zurich, Zurich, Switzerland
[Maffeis94]	A Flexible System Design to Support Object-Groups and Object-Oriented Distributed Programming, S. Maffeis, IFI TR 94.02, Dept. Of Computer Science, University Of Zurich, Zurich, Switzerland
[Makpangou91]	BOAR: A Library of Fragmented Object Types for Distributed Abstractions, M. Makpangou, Y. Gourhant, M. Shapiro, Proceedings Intl. Workshop on Object-orientation in Operating Systems, Paolo Alto, CA (U.S.A), October 1991
[Makpangou93]	Fragmented Object for Distributed Abstractions, M. Makpangou, Y. Gourhant, J-P Le Narzul, M. Shapiro, Readings In Distributed Computing Systems, IEEE Computer Society Press, July 1993
[Mullender90]	Amoeba - A Distributed Operating System for the 1990's, S. Mullender, G. van Rossum, A. Tannenbaum, R. van Renesse, H. van Staveren, IEEE Computer Magazine, May 1990
[Newman92]	Implicit Locking in the Ensemble Concurrent Object-Oriented Graphics Editor, R. Newman-Wolfe, M. Webb, M. Montes, CSCW '92 Proceedings, Toronto, Canada November 1992
[NeXT93]	NeXT Distributed Objects, NeXTStep General Reference, Release 3.2, NeXT Computer Inc, 1993
[Notkin93]	Adding Implicit Invocation To Languages: Three Approaches, Lecture Notes In Computer Science, S. Nishio, A. Yonezawa, editors, Object Technologies for Advanced Software, Springer-Verlag 1993
[Peterson89]	Preserving and Using Context Information in Interprocess Communication, L. Peterson, N. Buchholz, R. Schlichting ACM Transactions on Computer Systems, Vol. 7, No. 3, August 1989.
[Reiter92]	Integrating Security in a Group Oriented Distributed System, M. Reiter, K. Birman, L. Gong, Proceedings of 1992 IEEE Symposium on Research in Security and Privacy, 1992

[Salzer84] End-to-end Arguments In System Design,
J.H. Salzer, D.P. Reed, D.D. Clark,
ACM Transactions on Computer Systems, Vol.2, No. 4,
November 1984

[Schneider90] Implementing Fault-tolerant Services Using The State Machine Approach: A Tutorial,
F. Schneider,
ACM Computing Surveys, Vol. 22, No. 4,
December 1990

[Shapiro86] Structure and Encapsulation in Distributed Systems: the Proxy Principle,
M. Shapiro,
Proceedings 6th Intl. Conference on Distributed Computing Systems,
IEEE, May 1986

[Shapiro91] SOS: An Object-Oriented Operating System - Assessment and Perspectives,
M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, C. Valot,
Computing Systems, Vol. 2, No. 4,
Fall 1989

[Shrivastava89] An Overview of the Arjuna Distributed Programming System,
S.K. Shrivastava, G.N. Dixon, G.D. Parrington
Technical Report, Computing Laboratory, University of Newcastle-upon-Tyne,
Newcastle-upon-Tyne, England

[Skarra86] An Object Server for an Object-oriented Database System,
A. Skarra, S. Zdonik, S. Reiss,
Proceedings of International Workshop on Object-oriented Database Systems,
Sept. 1986

[Spector88] Synchronizing Shared Abstract Types,
P. Schwarz, A. Spector,
ACM Transactions on Computer Systems, Vol. 2, No. 3,
August 1984

[TEK93] Teknekron Information Bus: Programmer's Reference Manual,
Teknekron Software Systems,
Palo Alto, California, May 1992

[Thomas79] A Majority Consensus Approach to Concurrency Control for Multiple CopyDatabases,
R. Thomas,
ACM Transactions on Database Systems, Vol. 4, No. 2,
June 1979

[Tichy82] Design, Implementation and Evaluation of A Revision Control System,
Proceedings of the 6th Intl. Conf. on Software Engineering,
IEEE, New York,
1982

[vanRenesse93] The Horus System,
R. van Renesse, K. Birman, R. Cooper, B. Glade, P. Stephenson,
Technical Report, Cornell University
Ithaca, New York,
July 1993