Design and Implementation of CoObjects (a Shared Object Server for Cooperative Applications)

Ovidiu Sandor
(ovidiu@nada.kth.se)

The Royal Institute of Technology
Stockholm, Sweden
Abstract

For decades the computers have been designed as an individual working tool. Today the importance of all this technology tends to switch towards using the computer as the ultimate collaboration tool. Such Computer Supported Collaborative Work systems do and will need a special kind of infrastructure. In such a system distribution of data is central.

This report presents a project that tries to design, implement and use a Shared Object Server (SOS) module called CoObjects. The paper starts by defining the requirements of such a SOS based on the demands of cooperative applications. Then a model is suggested that fits the requirements. Later on, implementation problems are presented as well as the solutions adopted.

The resulting SOS is an object-oriented system that provides in an transparent way for the application writer the distribution of data and the optional persistent storage of it. The CoObjects SOS is a tightly-coupled distributed system with partial or total replicated objects. Replication is done by state transfer using total ordered group broadcast implemented with multicast addresses.

The SOS is build in a modular way making changes simple and allowing later additions. The implementation is quick, light, free, easy to install and use. The report describes the way the SOS has been used in the design of some cooperative applications. The paper concludes by highlighting the characteristics of the SOS as well as by suggesting future work on it.

Keywords: CSCW, Groupware, Object Oriented Programming, Distributed Systems, Replication, Concurrency Control, Object Oriented Databases
Acknowledgments

First of all, my thanks go to my adviser, Prof. Yngve Sundblad for introducing me to the world of Computer Supported Collaborative Work, and for the permanent guidance and encouragement during my work. His long experience in this field and his wide perspective on humans and their computers has been and will be a great source of inspiration for me.

I want to thank my colleague Konrad Tollmar for the frequent critique and valuable comments that he offered me. He was the one to suggest this topic to me.

Last but not least, I wish to thank all my colleagues at IPLab and NADA for the great research environment that they provide. It is a pleasure to work in this laboratory and in this department.

An earlier version of this paper has been presented as a thesis work at the Technical University of Timisoara, Romania.
# Table of Content

1. Introduction........................................................................................................... 9  
  1.1 Computer-Supported Cooperative Work......................................................... 9  
  1.2 The Collaborative Desktop Project................................................................. 10  
  1.3 Shared Object Server...................................................................................... 11  
  1.4 Related work..................................................................................................... 12  

2. SOS Requirements ............................................................................................... 13  
  2.1 CSCW Applications......................................................................................... 13  
  2.2 Distributed systems ......................................................................................... 15  
    2.2.1 Models of Distributed Computation.......................................................... 16  
    2.2.2 Synchronous vs. Asynchronous Systems................................................... 16  
    2.2.3 Failures...................................................................................................... 17  
  2.3 Distribution in the CSCW applications............................................................. 18  
    2.3.1 Replicated vs. centralized architecture...................................................... 18  
    2.3.2 The bandwidth of the communication....................................................... 19  
    2.3.3 Transparency of the distribution............................................................... 19  
    2.3.4 Asynchronous distribution...................................................................... 20  
    2.3.5 Object Distribution.................................................................................. 20  
  2.4 Concurrency Control ....................................................................................... 20  
    2.4.1 Atomic Commitment.................................................................................. 22  
    2.4.2 Ordering..................................................................................................... 23  
  2.5 Persistent Storage ........................................................................................... 27  

3 Implementing CoObjects ...................................................................................... 29  
  3.1 The Tools .......................................................................................................... 29  
    3.1.1 C++ ........................................................................................................... 29  
    3.1.2 SID ............................................................................................................ 29  
    3.1.3 Multi-threads ............................................................................................. 30  
    3.1.4 Ode ............................................................................................................ 31  
  3.2 The Model ......................................................................................................... 31  
    3.2.1 Objects, Names, Managers and Adapters .................................................. 31  
    3.2.2 The General Structure.............................................................................. 31  
    3.2.3 Event Manager.......................................................................................... 32  
    3.2.4 Replication Manager................................................................................ 33  
    3.2.5 Name Manager and Name Adapters.......................................................... 34  
    3.2.6 Session Manager....................................................................................... 34  
    3.2.6 BasicSharedObject.................................................................................... 34  
    3.2.7 The Persistent Storage Process................................................................. 39  
    3.2.8 Defining new Object types...................................................................... 40  
    3.2.9 Conference control.................................................................................... 41  

4. Using CoObjects .................................................................................................. 43  
  4.1 @work .............................................................................................................. 43  
  4.2 The new CoDesk ............................................................................................. 46  

5. Future work: .................................................................
1. Introduction

1.1 Computer-Supported Cooperative Work

The basic difference between humans and animals is the fact that people can work together. Studies have shown that certain animals can bring up their babies (like the penguins), can use tools (like the chimps) or even communicate (like the dolphins). Additionally to all that the humans can work together. This brought our society as far as it is today. You can hardly find nowadays an invention or a discovery that has been done by a single person. Most of them are the result of the creativity and the effort of a howl group.

Understanding the role of the collaboration in the day by day life, people have started to think about tools that could make their collaboration easier. Even the creation of languages and alphabets must be seen as an effort to make collaboration easier. Close to our time people invented the telegraph and the telephone in order to make communication part of the daily life. This century brought us television, telex, fax, beepers and mobile phones are redefining the way we communicate, the way we work and the relations between humans.

In the last 20 years information technology changed the way we work. Now almost each office has a computer, a network connection and some programs helping people to reach goals easier and quicker then before. Still the howl hardware and software in use today does not take in consideration the need for human collaboration.

After some early experiences in the 60's, among which Douglas Engelbart's AUGMENT is the notable one, in the beginning of the 80's more and more research has been carried out in order to create a computer-based technology that should support people's collaboration. In 1984 Irene Greif invented the term Computer-Supported Cooperative Work (CSCW) for this new field of study. Since then many definitions have been formulated for CSCW:

"CSCW is computer-assisted coordinated activity such as problem solving and communication carried out by a group of collaborating individuals" [Greif88]

Another definition, adopted by many researchers, is the one suggested by Bannon and Schmidt:

"CSCW should be conceived as an endeavor to understand the nature and characteristics of cooperative work with the objective of designing adequate computer-based technologies for cooperative work arrangements.” [Bannon92]

Making the definitions more precise, CSCW focuses on the design and implementation of cooperative systems but also on developing the basic understanding of the underlying principals of group work, group behavior, etc. As a field of study, CSCW must be understand as an interdisciplinary research, involving computer scientists as well as researchers from the areas of psychology, sociology, organizational science, language theory, cognitive science, anthropology.
Groupware is the term used to identify the CSCW software, in contrast to the singleware, the normal single-user software. A good definition could be:

"Groupware is distinguished from normal software by the basic assumption it makes: groupware makes the user aware that he is part of a group, while most other software seeks to hide and protect users from each other....Groupware ...is software that accentuates the multiple user environment, coordinating and orchestrating things so that users can 'see' each other, yet do not conflict with each other" [Lynch90]

Nowadays a big research effort is put in this field both in Universities and in the software industry. Video/audio conferencing, multi-user editors, different other communication tools have become a reality. In the future CSCW will become the day by day reality for millions of people all over the world.

1.2 The Collaborative Desktop Project

The research described in this paper is part of an ongoing project called The Collaborative Desktop (CoDesk). The project is part of the COMIC Project, Esprit Basic Research Action 6225, of the European Community [COMIC94] [http://www.lancs.ac.uk/computing/research/soft_eng/comic/] and of the Swedish MultiG Project.

The Collaborative Desktop [Marmolin94] [Sundblad95] [Avatare92], consists of a set of generic tools for CSCW. The Collaborative Desktop is an attempt to make collaboration a natural part of the daily use of a computer. Our way to achieve this is to put the user in the center of the computing in a similar way that applications and documents are defined and visualized in the Apple Macintosh Finder metaphor of the daily-work desktop.

The KnowledgeNet [Marmolin91a] [Marmolin91b] is a vision of a system for collaboration in teams where the members have access to a common base of information, including knowledge about who-knows-what. The design of CoDesk is based on its function as an interface to TheKnowledgeNet.

CoDesk, the Collaborative Desktop (Figure 1.1), tries to integrate the essence in communication and collaboration via different media and tools such as text, graphics, shared windows, sound and video. The members are the prime objects in a window based direct manipulative interface, in which the members can form their collaboration environments based on a room metaphor. In this context the room metaphor is chosen to decrease the social and physical distance between the members of the cooperating groups.

The prototype of the Collaborative Desktop is having a pictorial, icon-based, directly manipulated graphical user interface that strives to:

- give a more seamless integration of synchronous and asynchronous modes of collaboration, a social ad-hoc communication to complete more formal and planned work processes and ability to determine and control who is communicating with who at a given time;
- be an understandable user interface that make computer mediated cooperation and collaboration an natural part of daily work;
- give awareness information about the group and the members activities in order to increase interaction and collaboration.

The prototype includes tools for handling team maps (including information on who-knows-what), connection (telephone and video) exchange, answering
machines, common white boards, collaboration in writing documents, mail and bulletin board tools.

Additionally, specific application for communication and collaboration have been designed and implemented. These include CoMail, CoBoard, CoEdit, MultiDraw. CoBoard is an electronic board where people can "stick" post-it notes. Other people can read them, edit them, etc. CoEdit is a multi-user text editor that allows synchronous editing by users placed on different sites. MultiDraw is a synchronous multi-user drawing editor.

1.3 Shared Object Server

It is clear that such a system and all the other CSCW applications need distribution of data as well as the persistent storage of it.

This paper will focus on the design and implementation of a program module that should solve most of the problems of the data distribution and of the persistent storage of it. We will call this module Shared Object Server, term suggested by other COMIC researchers [COMIC94].

The goal of the research described here was to study the requirements of CSCW applications in terms of underlying technology. Aspects of distribution of data and services should be carefully considered as well as data storage.

The project should use these requirements for defining a model of a Shared Object Server specialized for CSCW applications. The model should be implemented according to the specifications and requirements providing in this way a program module that should be used in future CSCW applications development in the laboratory. Finally applications should be build using the program module and the results should be evaluated.
The SOS should respect some basic requirements. Later on we will define the requirements that are typical for CSCW applications. Basic requirements include:

- SOS must be reliable, that is fault tolerant
- SOS must be quick
- SOS must be light
- SOS must be easy to be integrated in applications, easy to be installed and to be used
- SOS must be easy to be modified and to be improved
- SOS must be very cheap for the user, if possible for free, as well as any package or system included in it

The basic idea must remain the one that:
- SOS must offer a simple, transparent solution to the problems of distribution of data in CSCW applications

Although SOS will be designed having CSCW applications in mind, the SOS can be a general propose distribution package.

### 1.4 Related work

In the framework of COMIC some European Universities are undergoing studies in the design and implementation of SOS and SIS (Shared Interface Server) systems. The most notable ones are COLA and GroupDesk.

**COLA** is a object distribution system from the University of Lancaster that uses relations among objects as well as events for object distribution.

**GroupDesk** is a SOS developed on top of CORBA. This system uses events for data distribution and the CORBA build-in persistent storage mechanism.

**CORBA (Common Object Request Broker Architecture) of the OMG (Object Management Group)** is a big project involving over 200 software companies. CORBA tries to define an object oriented system that should provide distributed object access using a simple and uniform syntax. The system will be platform independent as well as language independent. The specifications of such a system are ready and the first implementations are on their way. SunSoft is in beta-testing with a UNIX operating system based on the CORBA specifications.

Unfortunately CORBA has some problems that makes it unfit for our goal. First, the lack of complete implementations. Second any CORBA system is big and complex making everything slow. Still, the biggest problem of using CORBA in CSCW applications is the fact that it uses a client-server architecture.

Another object distribution system is Arjuna [Little93] created at the University of Newcastle upon Tyne. The system uses atomic transactions for consistency control.
2. SOS Requirements

2.1 CSCW Applications

In order to define the requirements of the SOS for CSCW applications we have to see first what makes groupware different from singleware. For pointing out the difference I will use the Model-View architecture of an application.

A singleware application (a graphical editor, for example) has a structure like the one in Figure 2.1. The Model contains the data that is edited. The View contains the representation of the Model that is fit to be displayed to the user. Normally one data from the Model can have more than one representations in the View. The representation in the View adds to the data all the information needed for displaying and interacting with it (position, state, etc.).

![Figure 2.1. The Model-View architecture of singleware.](image)

The data can be modified by the action of the user on the view. The View will pass the event over to the Model. Here the data will be changed and then all the views of that data will be updated in order to be consistent with it. In this way the result of the action will be visible to the user.

In the case of groupware the structure becomes more complex, like in Figure 2.2. In addition to the architecture of singleware the problem of interacting with other users appears. When the user takes an action this is passed over by the View to the Model. After modifying the data all views of it will be updated. Additionally all other users are updated. When other users change data the Model is notified and after modifying the data all views will be updated.
In order to analyze the needs of CSCW applications we will use the model suggested by Ellis [Ellis91] called groupware time-space matrix (Figure 2.3).

This matrix divide the interaction in CSCW applications in synchronous or asynchronous and the users as local or remote. A CSCW application will try to support one or more of these kinds of interaction. These characteristics will have an effect on the requirements and properties of the applications.

The "face to face" and the asynchronous interactions are harder to support electronically. Normally a white-board solves the first interaction while a bulletin-board or "post-it" notes solve the second. Normally groupware tries to solve the distributed kinds of interaction. This project will focus on these types of applications.

From the functional point of view the structure of a CSCW application looks like in Figure 2.4.
The SOS fits in this structure like in Figure 2.5. The SOS will hide all the details related to the distribution and persistent storage of the data in the application.

2.2 Distributed systems

We should start this section by giving some definitions. First what is a distributed system? Intuitively a distributed system is several computers doing something together. Putting this in a more structured way we can define [Schroeder93] the following important components of a distributed system:

- **multiple computers** - each of them having it's own CPU(s), memory, as well as persistent storage devices (like disks)
- **interconnections** - some physical connections existing between the computers, allowing them to communicate (like wires, waves, etc.)
- **shared state** - the fact that "the computers are doing something together" can be formalized to the fact that the computers try to maintain some shared state.

Historically speaking distributed systems (in one form or another) have been present since the 60's. During time the following types of system have been used (and some still are) [Birman94]:

- **time-shared mainframe computer systems** - this was the first way for offering multi-user access to the computer mainframes of the 60's and 70's. The problem of these operating systems was to provide methods for correct running in such an environment (like synchronization, concurrency, virtual memory, etc.)
- **networked time-shared mainframes** - the first networked mainframes and minicomputers focused on applications like file transfer, remote login, electronic mail and bulletin board services. In this period some studies of distributed systems have been carried out. Problems like network management, packet routing, error control and error correction have been solved.
- **client-server workstation architecture** - in the 80's it became possible to build workstations with the power of the mainframes. The new face of networking
became rings of clients around servers that are managing files, databases and services. In these systems new ideas have been tested (like window-based user interfaces, desktop publishing, etc.). Also some work has been put in the distribution problems (like distributed access to databases). Everything in this architecture happens by requests from clients and replies from servers.

- **closely coupled distributed computing systems** - these are systems where the components cooperate closely, sharing important states and announcing changes in the system.

### 2.2.1 Models of Distributed Computation

In order to study a distributed system we will redefine its definition by replacing the computer with processes. This gives a more general definition. Now we will have in our system processes (located on different computers, possibly more then one process on one computer), interconnections and shared states.

Distributed systems have been studied using different computational models.

A first criterion splits them in *message-passing* and *shared memory* models. In the former the processes communicate by sending and receiving messages over communication links. In the later the processes communicate by accessing shared objects (like registers, queues, etc.) We used the first model as we found it closer to the intuitive and technical way things work in nowadays networking systems.

### 2.2.2 Synchronous vs. Asynchronous Systems

When modeling distributed systems it is useful to make a difference between *synchronous* and *asynchronous* systems. Unfortunately the terms are the same with the ones defining interaction. Still these terms describe different things.

In the more general case, the asynchronous systems, we make no assumption about the message delivery delay or about the process execution speed. Saying that a system is asynchronous is a non-assumption. All systems are asynchronous as they all satisfy the definition. Protocols defined for asynchronous systems can be used in any distributed system. Unfortunately in asynchronous systems certain problems cannot be solved. The most famous example for that is the **Coordination Problem**: Two processes A and B communicate by sending and receiving messages on a bi-directional link. Neither process can fail. However the link can exhibit transient failures resulting in the loss of a subset of the messages that have been sent. Devise a protocol where either of two actions x or y are possible, but both processes take the same action and neither take both actions.

As mentioned, this problem has no solution in an asynchronous system. Here follows the demonstration:

In order to prove this we assume that a protocol exists and then we will try to find a contradiction. Assuming that a protocol that solves the problem exists, then such a protocol is equivalent to one in which we have rounds of messages exchanged: A sends a message to B, then B sends a message to A, and so on. From all the equivalent protocols we choose the one with the fewest rounds. According to our assumptions no protocol that solves the problem with fewer rounds exists.

Suppose that m is the last message and that it is sent by A. Because A cannot find out if B receives m we can say that the action taken by A doesn't depend on m. At the other end the action taken by B cannot depend on m because m may be lost (because of link failure).
As neither A nor B depends on m we can say that m is superfluous and we can remove it. In this way we constructed a protocol that uses fewer rounds to solve the problem. But the existence of such a protocol contradicts the assumption that the original protocol used the fewest numbers of rounds.

A synchronous system is a system where bounds are defined for message delivery delay as well as on process execution speed. Formally we can define as synchronous a system with the following properties:

• there exists a known upper bound on message delivery delay
• there exist known upper and lower bounds on the time required by a process to execute a step

Because of these bounds protocols can detect failure based on time-outs. In this way new methods can be implemented where time-based techniques solve problems that are impossible to solve in asynchronous systems.

2.2.3 Failures

In a general distributed system we have four major problems to solve [Schroeder93]:

• independent failure - as we have several distinct processes we would like to have the system still working correctly even if one or more of them have failed.
• unreliable communication - sometimes the connections between the computers do not work correctly; connections may not be available, messages can get lost or garbled. In this situation two processes may not be able to communicate even if both work properly.
• insecure communication - connections between processes may be exposed to unauthorized listening and message modification.
• costly communication - normally connections provide lower bandwidth, higher latency and higher communication cost than single machine applications.

Any general distributed system must take all this in consideration from the initial stage till the final development. Specific systems will emphasize one or some of these problems according to the requirements and the conditions in which the system is supposed to work.

If we consider the problem of independent failure we can include the following failure models commonly found in the distributed systems literature, as collected in [Schneider94]:

• failstop - a process fails by halting. Once this happened the process remains in this state. The halted state of the process can be detected by the other processes.
• crash - a process fails by halting. Once this happened the process remains in this state. The halted state of the process may not be detected by the other processes.
• crash+link - a process fails by halting. Once this happened the process remains in this state. A link (connection) fails by losing some messages but does not delay, duplicate or corrupt messages.
• receive omission - a process fails by receiving a subset of the messages that have been sent to it or by halting and remaining halted.
• send omission - a process fails by transmitting only a subset of the messages that it actually attempts to send or by halting and remaining halted.
• *general omission* - a process fails by receiving a subset of the messages that have been sent to it, by transmitting only a subset of the messages that it actually attempts to send or by halting and remaining halted.

• *Byzantine failure* - a process fails by exhibiting arbitrary behavior.

For the problem of communication failures we can have the following models [Hadzilacos93]:

• *crash* - a link fails by stopping transporting messages. Before stopping, however, it behaves correctly.

• *omission* - a link fails by transporting only a subset of the messages it is supposed to.

• *Byzantine* - a link fails by exhibiting arbitrary behavior.

These models of failure are used during the design and analysis of distributed systems. Even if not all real systems have all of these behaviors (like the Byzantine ones, for example) it is important to take them in consideration.

### 2.3 Distribution in the CSCW applications

Distribution has two major characteristics:

• the degree of coupling of the distributed system

• the bandwidth of the communication

The degree of coupling can vary greatly from one system to another. The extremes are full autonomous and close coupled systems.

For example email systems are autonomous. They can provide asynchronous distributed interaction based on text messages. Resource sharing systems offer both a good degree of coupling as well as autonomy to the sites. UNIX applications (rlogin, ftp, etc.) fall in this kind of systems. Distributed operating systems require more coupling. Still these systems are based on client-server architecture.

In the last time some systems have been developed that provide an interaction called group interaction in which the system is close coupled. These systems switch from the server-client architecture to a more distributed one, the replicated architecture.

### 2.3.1 Replicated vs. centralized architecture

Researchers have long argued the merits of replicated vs. centralized architecture. At the first site the centralized systems seem simpler to implement. Advantages of such a system include easy synchronization, default consistent states, easy locking, etc. Problems with such an architecture are latency (speed) and fault-tolerance. Such a system will fail in the case of the failing on the server, thought all other processes are alive. Increasing the fault-tolerance (with backup techniques) brings additional problems. On the other hand, replicated architecture raise problems of concurrency control. Such an architecture provides a much better fault-tolerance and a bigger flexibility. Different sites are more aware of the other ones compared to the centralized system.

For the SOS we decided on a replicated architecture that fits better the needs of CSCW applications, especially of the synchronous ones.
2.3.2 The bandwidth of the communication

As this aspect is concerned, CSCW applications need the maximum physical possible. This is compulsory in order to support multimedia elements (like video, sound, etc.). SOS must use such communication systems and protocols that allow the best bandwidth. SOS based applications will work in the Internet. Considering this it is clear that short distance communication will provide a better bandwidth compared to long distance communication where physical limitations appear. It is clear that the selection of the communication method will be of great importance for the performances of the SOS.

2.3.3 Transparency of the distribution

The different aspects of the distribution can be visible or invisible for the user. In the case of the SOS we consider the application writer as the user. The degree of control that he has over the distribution defines the transparency of the SOS. We can identify the following aspects of the distribution and the transparency effect on them:

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Central issue</th>
<th>Result of transparency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>The location of an object in the distributed system</td>
<td>User is unaware of the location of the object</td>
</tr>
<tr>
<td>Access</td>
<td>The method of accessing the object</td>
<td>All object are accessed in the same way</td>
</tr>
<tr>
<td>Migration</td>
<td>The re-location of an object</td>
<td>Objects may move without the user been aware</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Shared access to objects</td>
<td>Users do not have to deal with this problem</td>
</tr>
<tr>
<td>Replication</td>
<td>Maintaining consistent copies of replicated objects</td>
<td>Users do not have to deal with this problem</td>
</tr>
<tr>
<td>Failure</td>
<td>Partial falter in the system</td>
<td>Such problems are masked from the user</td>
</tr>
</tbody>
</table>

The aspects that should be transparent in the SOS must be defined. The problem is the one of making the use of the SOS simple (transparent) but still leaving the liberty to the user to control important aspects of the distribution. For SOS the requirements of transparency are:

- Location - transparent; as SOS is a replicated system each objects has a copy on each site so each site uses the local copy.
- Access - transparent; SOS must provide a uniform syntax for accessing all distributed objects.
- Migration - transparent.
- Replication - transparent.
- Failure - transparent with optional notification in order to allow the user to react on partial failures.
- Concurrency control - transparent (implicit method) but with the possibility of user redefining it.

Each of this problems will be detailed later on as well as the solution suggested.
2.3.4 Asynchronous distribution

In the case of asynchronous distributed applications another problem appears: storing the data for later use. The solution of having a server that keeps in memory all the data till later is not usable because reuse can appear days, months or years later. The probability of such servers to fail meantime is to high.

The only solution is the persistent storage of data on magnetic supports. This can include storage in file systems, in databases, etc. This defines a new requirement:

- The SOS must provide the functionality for the optional persistent storage of data

Still the SOS must remain consistent with the transparency previously defined. It is clear that access of persistent data must be identical with any other access.

2.3.5 Object Distribution

An object can be formally split in object state and object behavior. Practically the object state are the member variables while object behavior are member functions.

In an ideal OO replicated system we would desire to distribute howl objects by sending them from one site to the other. Unfortunately C++ makes such an attempt impossible. This for several reasons.

The first problem that would appear would be the one of sending member functions over the network. The point is that C++ uses virtual function tables. How would such tables be created on a remote site when certain functions would be transferred? The memory image of both the functions and the virtual tables are implementation dependent making there distribution over heterogeneous systems more complicated.

Second, even if function transfer could be possible, doing that would require a violation of object encapsulation. To have the functions execute at the remote site private data members must be accessed.

Taking all this in consideration certain distributed systems decided either to provide distributed object state or distributed object behavior. Distributing the object behavior is closer to the client-server architecture where a client requests an action (activity, behavior) from the server. The distribution of object state is closer to replicated architecture where a shared state is kept on all sites based on announcing changes in the local state. Our previous experience with CoDesk shows that shared state is the biggest need in CSCW real-time applications and environments. Anyway, the point is that even if one of the distribution strategies is chosen, the other one can be simulated. For example, in a distributed object state system the modification of a certain state variable could trigger a certain action at one or all sites. In the other case, of the distributed object behavior, parameters of the request and of the reply can be used in order to pass states between sites.

The CORBA project has decided to provide distributed behavior. The basic reasons for doing so where data protection and client-server architecture.

In our case we decided that the distribution of object state would fit better our goals.

2.4 Concurrency Control

One of the biggest problems in groupware is concurrency control. Groupware must be seen as real-time distributed systems used by more then one user at the same time. Normally the users can access the same set of data. If the interaction between
the users and the data happens without any concurrency control results can lead to inconsistency of data, of data representation and of the group's mental model.

In order to represent the problems of concurrency control we will imagine a multi-user WYSIWYG graphics editor. Two, or more users try to draw something together. Let's consider that the users have a circle on there screen. Now user A coloreds the circle red. His application colors the circle and then sends that event to all other sites. In the (almost) same time user B colors the circle blue. His application colors the circle blue and then sends the event to all other sites. After some (short) time application B receives the event from application A that the circle has been colored red. So this will happen and user B will have a red circle on his screen. Meanwhile application A gets the event from application B to color the circle blue. In this way user A will see a red circle. Figure 2-1 shows the time-ordered events presented in this example.

Different methods have been suggested for solving this problem. Most of them enter the following categories:

- **Ordering (or serialization)**
- **Locking**
- **Atomic commitment (transactions)**

**Ordering** has basically two versions: non-optimistic or optimistic ordering.

In the **non-optimistic ordering** the system ensures that events are executed in the same order at all sites. Normally in this case a form of Total Ordering is used. The cost of such a method is the delay time of the events. This can make the system slow.

**Optimistic ordering** is based on the assumption that conflicting events are rarely received out of order. In this case a more relaxed ordering (and more quick) is used. This will increase the speed of the system. The consistency will be kept by regularly checking. In case certain events have been received out of order the consistency will be repaired. One method to do this is to roll back the howl system to the point where the inconsistency appeared and re-execute the events in the proper order. If operations are invertable then undo operations can be applied till the system reaches a consistent state.
**Locking** methods are based on mutual exclusion locks on objects. The first one to ask for a lock will get it and can use and modify an object as long as needed. After that the lock is released. If other users/processes want to use the object they have to wait for the lock to be released. Again optimistic and non-optimistic versions can be used.

In the *non-optimistic locking* method an object cannot be used and changed is the user/process doesn’t hold the lock on it. In the *optimistic* version the requester starts using the object and changes it. Later if the lock is granted everything is OK. Otherwise the object must be returned in the original state.

Unfortunately implementing a locking method depends on the data that is distributed. As the SOS is just a module and the real data to be distributed will be defined by the user there is no way to make locking efficient in such a general SOS. Problems related to granularity of locking, deadlock avoiding or detection would remain on the shoulders of the user of the SOS. Also delay in getting the lock would be undesired in interactive CSCW applications. Taking all this in consideration locking methods are not fit for the SOS.

### 2.4.1 Atomic Commitment

*Atomic commitment* (transactions) are techniques normally used in database systems. These are improved versions of the locking methods where the user has no concern about locking acquiring, locking granulation or deadlock detection. Such systems use complex algorithms, their implementation being usually slow and message demanding. A more in-depth discussion on these methods will follow in the paragraph on databases.

In a distributed system situations can appear where the execution of a certain sequence of events if needed, ensuring a that the action terminates consistently at all participating sites even in the presence of failures.

This kind of behavior is required especially from distributed database systems. The traditional example is a distributed bank account system. A client makes a payment from his account to the account of a company. The desired behavior is to have the amount withdrawn from the account of the client and the same amount registered in the account of the company. A possible scenario is that the amount of money is withdrawn from the account of the client but (because of failure) that amount is not registered in the account of the company. What we need is a protocol that guarantees that either both actions are taken at all sites or no action is taken at no site. This is the *Atomic Commitment Problem*.

Algorithms for solving the problem exist. Probably the best known is the Two-Phase Commit algorithm (2PHC). Unfortunately such an algorithm may result in blocking actions. In such a situation the ongoing transactions are aborted. The programmer must design his software in such a manner to consider deadlocking.

Modern algorithms have solved the blocking. Such a system is the Three-Phase Commit algorithm (3PHC). Such methods are complex and much more costly.

In both cases distribution must be understand as multiple users accessing one server. In the case of fully distributed databases (multiple users accessing multiple servers) the problem of atomic commitment gets more and more complicated.

The result of this situation is that we have to mix the atomic commitment access offered by the database with the concurrency control that we have chosen (ordering).
2.4.2 Ordering

Lamport was the one to introduce the idea of ordering of the events in distributed systems [Lamport78a] [Lamport78b]. We will use Lamport’s model and formalism for the discussion.

In the message-passing model we have two basic techniques: peer-to-peer messages and broadcast messages. In the first case a message is sent by one process with a well-defined receiving process. The message will be received by at most one process (it may be lost, thought). In the case of broadcasting, a message is sent by one process and all other processes can receive it. In this case the message can be received by at most $n$ processes where $n$ is the number of processes in the system.

We will talk first about ordering in peer-to-peer messages systems and then about ordering in broadcast messages systems.

Peer-to-peer

We will consider a distributed system as being a collection of processes. A process is defined as a sequence of events. We will say that an event $e$ happens before $e'$ if $e$ happens at an earlier time than $e'$. This is actually according to the physical meaning of time. In a process events form a sequence, where $e$ appears before $e'$ in the sequence if $e$ happens before $e'$. Considering this, a process can be seen as a set of events with an a priori total ordering. We can define a local history of process $p_i$ as being a (possibly infinite) sequence of events $h_i = e_i^1 e_i^2 ...$, where $e_i^1$ is the first event executed, $e_i^2$ is the second one, etc.

Processes send messages over channels. We consider a channel to be a reliable link that is any sent message will be delivered once and only once and no unsent messages will be ever delivered.

We will consider that sending or receiving a message is an event. Formal we will represent this by $e_i = \text{send}(m)$ or $e_i = \text{receive}(m)$.

We can define a binary relation, denoted by $\rightarrow$, as follows:

1. If $e$ and $e'$ are events in the same process and $e$ appears before $e'$, then $e \rightarrow e'$.
2. If $e$ is the sending of a message by a process and $e'$ the receiving of the same message by another process, then $e \rightarrow e'$.
3. If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

The same definition in a more formal way would be:

1. If $e_i^k, e_i^l \in h_i$ and $k < l$, then $e_i^k \rightarrow e_i^l$.
2. If $e_i = \text{send}(m)$ and $e_k = \text{receive}(m)$, then $e_i \rightarrow e_k$.
3. If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

The relation just defined is called the "causal relation". We can say that $e \rightarrow e'$ if and only if $e$ causally precedes $e'$. The only conclusion that can be drawn from $e \rightarrow e'$ is that the occurrence of $e'$ and its outcome may have been influenced by event $e$.

We can see that the causal relation is a partial ordering relation for the set of all events of the system. Because $e \rightarrow e$ has no physical meaning (there is no system in
that an event can appear before itself), we can add the ireflexible property to the causal relation. Certain events may be causally unrelated. That means that there can exist some \( e \) and \( e' \) for which neither \( e \rightarrow e' \) nor \( e' \rightarrow e \). Such events are called \textit{concurrent} and we denote them with \( e || e' \).

Now we will start ordering events in different processes by defining different delivery rules. Reordering is obtained by “halting” received messages till there turn in the order has arrived (as defined by the delivery rules). Then a \textit{deliver} primitive is called. From now on we will use the primitive \textit{deliver} instead of \textit{receive}, the later being hidden within the delivery rule.

The first rule that can be defined is the \textit{FIFO Delivery}:

\[
\text{For all } m \text{ and } m', \text{send}_i(m) \rightarrow \text{send}_j(m') \Rightarrow \text{deliver}_k(m) \rightarrow \text{deliver}_k(m')
\]

As we can see FIFO Delivery involves two processes \( p_i \) and \( p_k \) and the communication between them. In other words messages sent by one process arrive to the other one in the same order as they have been sent. The rule doesn't define any relation between event in other then the two processes.

Implementing FIFO Delivery over a non-FIFO channel is simple. The sending process adds a sequence number to its messages and the destination process presents the messages in the order of these sequence numbers.

A more powerful rule is the \textit{Causal Delivery}:

\[
\text{For all } m \text{ and } m', \text{send}_i(m) \rightarrow \text{send}_j(m') \Rightarrow \text{deliver}_k(m) \rightarrow \text{deliver}_k(m')
\]

The Causal Delivery extents ordering to all messages that are causally related even if they are sent by different processes. FIFO Delivery is not sufficient to guarantee Causal Delivery. \textit{Figure 2-2} is an example where the FIFO Delivery is satisfied but the Causal Delivery for \( p_3 \) is not.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{image.png}
\caption{Example of FIFO Delivery that is not Causal}
\end{figure}

\textbf{Broadcast}

In a broadcast system the problems are similar but a bit more complicated. We will use a definition system proposed in [Hadzilacos93].

As we use \textit{send} and \textit{deliver} to formalize peer-to-peer connections we will use \textit{broadcast} and \textit{deliver} for broadcasting connections.

The first rule that can be defined is the \textit{Reliable Broadcast} defined by the following properties:
Validity: If a correct process broadcasts message \( m \), then all correct processes eventually deliver \( m \).

Agreement: If a correct process delivers a message \( m \), then all correct processes deliver \( m \).

Integrity: For any message \( m \), every correct process delivers \( m \) at most once, and only if some process broadcast \( m \).

The type of broadcast is called **FIFO Broadcast** and is defined as a Reliable Broadcast that satisfies also the following requirement:

**FIFO Order**: If a process broadcasts a message \( m \) before it broadcasts a message \( m' \), then no correct process delivers \( m' \) unless it has previously delivered \( m \).

We can see that the FIFO Broadcast requires that messages broadcasted by the same process must be delivered in the same order as the one of broadcasting.

A **Causal Broadcast** is a Reliable Broadcast that satisfies the following requirement:

**Causal Order**: If the broadcast of a message \( m \) causally precedes the broadcast of a message \( m' \), then no correct process delivers \( m' \) unless it has previously delivered \( m \).

As we have seen at the peer-to-peer systems Causal Broadcast doesn't impose any ordering among events that are not causal related.

For example, consider a replicated database where a bank account has the value $100. The owner deposits $20, so the process at the bank will broadcast the "add $20" message. At the same time the central bank computer broadcasts the adding of interest on the money from the bank accounts: "add 10% of the amount". As the two events are not causal related the two messages can be received by the other processes in different orders. The result would be that some processes would have as a new value $130 and others $132 (see Figure 2-3).

![Figure 2-3 - Example where Causal Broadcast creates inconsistency](image)

To solve such problems we must define a rule that garanties that all correct processes deliver the messages in the same order. This is the **Atomic Broadcast** and it is defined as a Reliable Broadcast that satisfies the following rule:

**Total Order**: If process \( p \) and \( q \) both deliver messages \( m \) and \( m' \), then \( p \) delivers \( m \) before \( m' \) if and only if \( q \) delivers \( m \) before \( m' \).
Atomic Broadcast doesn't require the messages to be delivered in FIFO order. This can create problems in the case of transient failure of processes. We therefore define the **FIFO Atomic Broadcast** which is a Reliable Broadcast that satisfies both the FIFO and the Total Order requirements. FIFO Atomic Broadcast is stronger than both FIFO Broadcast and Atomic Broadcast.

Still, FIFO Atomic Broadcast doesn't require Causal Order for the message delivery. Therefore we define the **Causal Atomic Broadcast** that is a Reliable Broadcast that fulfills both the Causal Order and the Total Order requirements. This type of broadcast is stronger than both Causal Broadcast and Atomic Broadcast. This is in fact the key broadcast mechanism used in the State Machine approach to fault-tolerant systems.

Till now time had no role in the broadcast. Similar to the synchronous model we can define an upper bound to the time past from the moment of the broadcast and the moment of delivery. In order to define this bound we can use two methods: real time (like measured by an external observer) or local time (as measured by the local clocks of the processes). Considering this we can define this property (called Δ-Timeliness) in two ways:

- **Real-Time Δ-Timeliness**: There is a known constant Δ such that if the broadcast of m is initiated at real-time t, no correct process delivers m after real-time t+Δ.

For the second definition we assume that each message has a timestamp ts(m) expressing the local time of the sender when m was broadcasted.

- **Local-Time Δ-Timeliness**: There is a known constant Δ such that if the broadcast of m is initiated at local time (of the sender) ts(m), no correct process p delivers m after local time (of p) ts(m)+Δ.

A broadcast that satisfies one of these properties is called a Timed Broadcast. These properties can be combined with the other ones and can generate new kinds of broadcasts.

Two definitions are needed because the results are different. For example in an asynchronous system no Reliable Broadcast algorithm can satisfy either Real-Time or Local-Time Δ-Timeliness. In a synchronous system Reliable Broadcast can be implemented with respect to Local-Time Δ-Timeliness. Different fault models can influence other combinations of these properties.

All the requirements presented till now do not set any restrictions on faulty processes. For example the Agreement requirement allows a faulty process to deliver a message that is never delivered by the correct processes. Such behavior can have bad results in systems like distributed databases. In order to solve these problems some properties have been strengthened by adding the uniformity characteristic. So we can define the following new properties:

- **Uniform Agreement**: If a process (whether correct or faulty) delivers a message m, then all correct processes deliver m.

- **Uniform Integrity**: For any message m, every process (whether correct or faulty) delivers m at most once, and only if some process broadcast m.

- **Uniform FIFO Order**: If a process broadcasts a message m before it broadcasts a message m', then no process (whether correct or faulty) delivers m' unless it has previously delivered m.
**Uniform Causal Order:** If the broadcast of a message \( m \) causally precedes the broadcast of a message \( m' \), then no process (whether correct or faulty) delivers \( m' \) unless it has previously delivered \( m \).

**Uniform Total Order:** If process \( p \) and \( q \) (whether correct or faulty) both deliver messages \( m \) and \( m' \), then \( p \) delivers \( m \) before \( m' \) if and only if \( q \) delivers \( m \) before \( m' \).

**Uniform Real-Time \( \Delta \)-Timeliness:** There is a known constant \( \Delta \) such that if the broadcast of \( m \) is initiated at real-time \( t \), no process (whether correct or faulty) delivers \( m \) after real-time \( t + \Delta \).

**Uniform Local-Time \( \Delta \)-Timeliness:** There is a known constant \( \Delta \) such that if the broadcast of \( m \) is initiated at local time (of the sender) \( ts(m) \), no process \( p \) (whether correct or faulty) delivers \( m \) after local time (of \( p \)) \( ts(m) + \Delta \).

Using the uniform definitions we can redefine the different broadcasts. For example a **Uniform Timed Reliable Broadcast** will we one the satisfies Validity, Uniform Agreement, Uniform Integrity and Uniform \( \Delta \)-Timeliness.

From the practical point of view the most important types of broadcast are the Uniform Timed Reliable Broadcast (used in Non-Blocking Atomic Commitment), the Uniform Timed Causal Broadcast and the Uniform Timed Causal Atomic Broadcast (used in State-Machine systems).

In this implementation we use the Uniform Timed Causal Atomic Broadcast.

### 2.5 Persistent Storage

As one of the SOS requirements states the need for persistent storage we had to investigate methods for solving this problem. Two major ways of doing this came in our attention: databases and file systems.

**File systems** provide functionality for creating, accessing, modifying and deleting information units called files. Data in files is user-defined. Systems like NFS or AFS allow distributed file storage. Different applications use files for storing information.

From the point of view of the SOS file systems raise some problems. First the fact that the data is unstructured. This would require software parts for writing/reading the data to/from the files. The second and major problem is the fact that file systems have build-in privileged access mechanisms. The goal of such mechanisms is to protect data of one user from the other one. This comes in contradiction with the goal of CSCW applications.

**Databases** are software systems that allow the storage of data in a more structured and controlled way. Nowadays distributed databases are existing making the work easier. Such systems allow multiple users/processes to access one single database server. In this way data is shared among different users.

In the last years a number of databases appeared that use as storage data unit objects. These databases are called object oriented databases. They provide natural semantic for creating, accessing and modifying objects in the database.

For the implementation of the SOS we decided to use distributed object oriented databases.
3 Implementing CoObjects

3.1 The Tools

Before describing the implementation we will present the used tools and the reasons why these tools have been used.

3.1.1 C++

From the beginning of the project object-oriented program was chosen for the implementation of the different prototypes.

OO programming offers a good technique for modeling systems. First it offers methods for modular programming, making prototyping easier. Replacing, redesigning and rewriting of different modules can be made during the refinement process without changing other parts of the system. Big programs can be managed in an easy way by using encapsulation and detail hiding. Code reusability is much greater then in the case of traditional programming languages. Inheritance, polymorphism and modularity are used for that.

In the recent years a lot of OO packages have been design for almost all programming areas. Graphical User Interface (GUI) tools tend to become more and more OO. Visual GUI builders, largely used today, especially for prototyping, normally generate code in an OO language. Recently OO databases have been design. They tend to replace traditional relational and hierarchical databases. Other components needed in the development of software tend to use OO techniques.

The major options for OO languages have been C++ and Smalltalk. Smalltalk is a powerful medium for OO programming. Smalltalk is a incremental compiler that is in the same time the environment for running the programs. Although Smalltalk is better then C++ there are some additional problems with it. First the Smalltalk environment (needed for running the programs, not only for compiling) is very big (in memory and on disk). Second the cost of a good implementation is very high not only for the application developer but also for the user. Because of this and because of previous experience with C++, the later has been chosen for the implementation.

3.1.2 SID

Another decision that had to be taken regarded the protocol/software used for networking messages.

As close coupled distribution is a must for the SOS we had to choose among Isis or SID.

Our previous experience with CoDesk involved a package called Isis [ISIS94]. Isis is one of the first packages based on group broadcast. It is a complex system, providing a lot of functionality. This makes the installation and use of this system complicated.
SID [Hagsand94] is another broadcast package developed at SICS (Swedish Institute for Computer Science). The broadcast is implemented using IP multicast addresses. In this way a message is sent to all other sites using one single package of data. This makes communication very fast and keeps the traffic on the network low. Additionally to broadcasting messages, SID allows RPC as well as multi-RPC. Another important qualities of SID (especially compared to Isis) are: SID is a free software, it is easy to install, easy to set up and simple to use. The only negative aspect of SID is the fact that it is a C based package. We would have preferred a OO (C++) package.

As a conclusion SID proved to be the best choice. It offers exactly the functionality needed without being big, slow or complicated as other systems that try to provide much more services.

3.1.3 Multi-threads

A traditional process has a single thread of control. A thread of control, or simply a thread, is a sequence of instructions executed in a program.

Traditionally, in order to obtain concurrent execution of code one has to use several processes that communicate (by shared memory, pipes, messages, etc.). Multi-threads solves the problem in another way.

A multi-threaded process is no longer one thread of control but it is associated with one or more threads. Threads in a program execute independently. There is no way to know how different thread's code interleave. Threads share the same code and the same data. Still, each thread has its own Program Counter and its own stack.

Because data is shared some new behavior appear in the case of multi-threads. For example if a thread opens a file then any other thread can use that file. Certain system calls will effect all threads (like exit) while others only the thread that called it. Programming with threads requires more care and discipline then traditional programming because there is no system protection between threads.

Thread packages include diverse synchronization facilities to allow threads to cooperate in accessing shared data. Mutual exclusion (mutex) locks, condition variables and semaphores are the normal facilities offered.

Using multi-threads has been a natural decision because of the following considerations. First threads offer good support for concurrent programming. This is useful in a system that is expected to accept different events from different inputs (other processes sending messages, keyboard/mouse events, database updates, etc.). Second threads are better then several processes because the context of a thread if much smaller then the content of a process. In this way context switching between threads is considerably cheaper then context switch for processes. A practical reason was the fact that the SID distribution package uses multi-threads so it was straight forward to use threads in our code too.

Some systems offer kernel support for threads. Normal UNIX systems do not offer such support but packages have been designed for allowing multi-thread programming even in this conditions. In the last time efforts have been made to standardize multi-thread packages. As a part of the IEEE POSIX standard (The Portable Operating System Interface for Computer Environments) POSIX.4a [POSIX92] defines the Threads Extension.

Finally the decision was to use the pthreads package based on QuickThreads, the package coming together with SID.
3.1.4 Ode

The next decision that had to be taken was the database system to be used. We considered Gemstone, ObjectStore and Ode. All three are distributed object oriented database systems.

Gemstone is a OO database initially designed for Smalltalk. Today a C++ interface exists. Unfortunately the fact that Smalltalk and C++ have different philosophies makes the C++ interface unnatural. Additionally the Gemstone system is big and expensive (and must be paid both by the application developer and by the user).

ObjectStore is a OO database designed for C++. Interfaces to C and Smalltalk exist too. Again the cost of the system made the use of it prohibitive to us.

Ode [Ode95] from AT&T Bell Laboratories provided good database support. It extends the C++ language in a new language called O++. Actually the major extension is the \texttt{persistent} keyword that defines a class type or an object that is to be stored in the database. Of course, atomic commitment control semantic has been also added (\texttt{trans}). Ode uses a blocking Two-Version Two-Phase Commit protocol (2V2PHC). Advanced facilities include naming of objects, versioning and triggers. Additionally a OdeNFS file extension exists that allows the access to objects in the database as normal files. Being provided free for research the Ode database was the natural choice.

3.2 The Model

"Inside each replicated system there is a non-replicated system trying to get out."

As a result of the requirements analysis that we made in the previous chapter, the SOS is a close coupled object replication system. Replication is obtained by object state transfer using total ordered group broadcast.

3.2.1 Objects, Names, Managers and Adapters

Objects are the elements that are to be distributed. They are defined by the user according to certain rules.

Each Object has a Name that allows processes to access it. Each application names each object created. All names are unique and independent from the location of the objects. In certain situations the name is irrelevant, case in which an arbitrary name is given to that Object.

The Managers are the parts of the SOS that control everything. The functionality is structured in the Managers according to classes of services. It will be normal to have a Manager for each major problem that must be solved: event distribution, replication, naming, etc. The SOS contains a certain number of Managers but the user can define additional ones. For doing that the user must respect certain rules and conditions.

Adapters are structures that provide a certain type of interface between different components.

3.2.2 The General Structure

\textit{Figure 3.1} shows the general structure of the SOS.
The Event Manager will solve all problems related to distributing events. The Replication Manager has the role to ensure that the local copies are consistent with the distributed ones. The Replication Manager will be called in the case of changes in any Object. He will call the Event Manager in order to distribute this change. The Name Manager has the role of keeping track of all local copies of the distributed Objects. Name Adapters have the role of binding the Objects with their names. The Session Manager keeps track of all existing Managers and helps starting up and stopping the SOS in the current process.

After receiving an event the Event Manager will use the Name Manager in order to locate the local copy of the Object that is to get the event. After finding it, the Event Manager passes the event to the destination Object. Here the event will be processed and, if needed, the view of that Object will be updated.

### 3.2.3 Event Manager

The Event Manager is responsible for all communication in the system. It also hides all the technical details from the other parts of the system. In this way the distribution package could be changed without major modifications in other parts of the system.

The Event Manager is the one that sends all the events from one site to all others. Any modification that the application (the user) makes on a certain object will be translated in one or more events that are going to be broadcasted to all other sites.

In order to pass information from one site to the other marshaling and unmarshaling functions are used. In our implementation we have chosen io-streams as the support for marshaling and unmarshaling data.

As we already said, the Event Manager uses the SID distribution package for distributing the data. The io-streams are converted in data needed for the SID and then distributed. In the other way around, the Event Manager receives events from other processes and then converts it to io-streams that are pasted to other managers or to objects.
Creating the Managers

Each application uses a certain number of Managers. The way they interact make it very difficult to decide when to create a Manager and when to destroy it. In order to avoid this problem we implemented a mechanism that allows the creation of Managers in the moment of the first request. A traditional way of using a Manager would look like:

```cpp
EventManager* em;
//...
em = new EventManager;
em->run();
```

The problem is that we use a single Event Manager and we do not know where it will be used for the first time. A solution would be to create all Managers at the beginning of the program but this creates other problems related to the order of initialization of variables and objects. Another problem is keeping the pointers to the Managers so that they can be accessed from all parts of the program. In order to solve this problem solutions could be to have global pointers to this Managers, to have a superclass containing all the pointers or keeping pointers to the needed Managers in all classes that access them.

The mechanism that we implemented uses the notion of instance. Each Manager exist as exactly one instance of its class type. The first request to access the instance will create the object and store it in the instance. All following requests for the instance will return the object already created. The code will look now like:

```cpp
EventManager* em;
//...
em = EventManager::instance();
em->run();
```

Implementing the instance function follows:

```cpp
class EventManager {
    //...
    public:
        static EventManager* instance();
    }

static EventManager* _em = NULL;
EventManager* EventManager::instance() {
    if (_em == NULL) {
        _em = new EventManager;
    }
    return _em;
}
```

This solution also solves the problem of keeping pointers to each Manager in different parts of the system.

### 3.2.4 Replication Manager

The Replication Manager is responsible for the replication of objects. It will use the Event Manager for communication. Usually the Replication Manager is called by an Object during his commit or by remote processes. For example, the Replication Manager takes care of creating and deleting Objects on remote sites.

The Replication Manager is also responsible for sending a complete copy of all Objects in the distributed system to any new joining process. This will be done by only one of the already existing processes.
Partial vs. total replication

The replication of the Objects can be done in two ways: total or partial. The difference is the number of Objects on which a process holds a local copy.

In the total replicated system all Objects have copies on all sites. Sometimes a process doesn’t need all these Objects. The result could be a partial replicated system. In this case no state transfer takes place at the joining of a process. A process will ask for a copy of an Object only if it needs it. Eventually the copy of the Object can be removed later. This technique raises some problems, for example the fact that at least one process must hold a copy of a certain Object otherwise the Object doesn’t exist anymore.

In the actual implementation we used the total replicated system. In the future effort will be put in implementing the partial replicated version.

3.2.5 Name Manager and Name Adapters

The Name Manager is responsible for mapping names to objects. The Name Manager should work as a dictionary where all objects are recorded. The Name Manager uses Name Adapters in order to interface to the Objects. From the implementation point of view, the Name Manager is a hash table.

3.2.6 Session Manager

The Session Manager is responsible for making the current process part of a CoObjects system. It also will create or refer Objects describing the process and the machine. Also a authentication of the user of the process is done by calling the Authorization Manager.

3.2.6 BasicSharedObject

The Objects in the system are to be defined by the user. The CoObjects contains a class called BasicSharedObject that must be subclassed in order to obtain distributed objects with optional persistent storage. The interface of this class follows:

```cpp
class BasicSharedObject{
   public:
      BasicSharedObject();
      virtual ~BasicSharedObject();
      ...
      virtual void name(ObjectName);
      virtual ObjectName name();
      virtual void unname();

      virtual void commit();
      virtual void changed(EventType);

      virtual void persist();
      virtual void unpersist();
      virtual int is_persist();

      virtual void set_function(CallbackFunction);
      virtual void unset_function();

      virtual int is_a(ObjectType t);
      virtual ObjectType object_type();

      virtual istream& set_value(istream&);
};
```
virtual ostream& get_value(ostream&);
...
};

The first set of functions are the ones related to the name of the object. The `name(ObjectName)` function will create a Name Adapter with the provided name, will bind this Object to it and then will add the name in the Name Manager. The `name()` function is used to access the name of an Object. The `unname()` function is used to remove an Object from the Name Manager.

The function `commit()` is called by the user whenever the Object has changed. This will generate a event that notifies all other copies of this Object that the value has changed. On the remote sites the function `changed()` will be called in the copy of the Object.

The functions `set_value()` and `get_value()` are used as marshaling and unmarshaling functions. They convert a istream in the values of the Object member variables respectively from these variables in an ostream.

The callback mechanism

The functions `set_function()` and `unset_function()` are used to create user defined callbacks for Object change events. For example if a user would like to execute some code in the case a certain Object is deleted (by the current process or by a remote one) he will set a callback function in that Object. On any event that function will be called. There the user can select the desired event and take the needed action. In the Model-View architecture this callback mechanism could be used for updating the View of an distributed Object.

The basic difference between the `changed()` function and the callback mechanism is the fact that the first one implements a behaviour of any Object of that class while the later allows a specific behaviour for each Object.

Persistent Objects

Any Object in the system has the possibility of becoming persistent. This is done by the `persist()`, `unpersist()` and `is_persist()` functions.

The first function defines the current Object as a persistent one. The effect of calling this function is that the Object will be stored persistent and all other copies of the Object will be announced about this. The `unpersist()` function has the invers effect: the Object will be deleted from the database and all other copies will be informed about this. Of course, the Object remains in the memory of all processes like any other normal distributed Object. The `is_persist()` function is an interrogative one and returns true is the current Object is persistent stored.

The exact way things work will be explained a bit later.

Downcasting of objects

One of the problems of C++ is downcasting [Stroustrup94].

Lets define the problem. Assuming that `Person` is a class:

```cpp
class Person {
    //...
public:
    //...
};
```
And a different subclasses of `Person`, among them the class `Student` that contains a new function `get_mark()`:

```cpp
class Student : public Person {
    //...
    public:
        int get_mark();
};
```

Now let us imagine that objects of type `Person` and subtypes (including `Student`) are stored in a list. The `List` class has a function that handles us a pointer to a `Person` object. How do I know if that is a `Student` type object and if I can call the function `get_mark()`?

The information lacking here is called sometimes Run-Time Type Information (RTTI). Different ways for solving this problem have been suggested. Still each one has problems.

The most important solution to this problem is the one accepted in 1993 as an extension to C++ by the ANSI/ISO C++ committee. The mechanism consists of three parts [Stroustrup94] [Stroustrup92]:

- an operator `dynamic_cast`
- an operator `typeid`
- a structure `type_info`

The operator `dynamic_cast` will get as a parameter a pointer to an object of the base class and will return a pointer to an object of the derived class only if the object pointed to is really of the specified class. Otherwise the return value will be 0.

The operator `typeid` returns the exact type of an object given a pointer to a base class. The `type_info` structure is used as a hook for further run-time information associated with a type.

The naive solution to our problem would look like this:

```cpp
List *list;
Person *p;
Student *s;
//...
p = list->get_next();
if (typeid(*p) == typeid(Student)) {
    s = (Student*) p;
    int m = s->get_mark();
    //...
} else {
    // p is not a Student
}
```

This seems to be a good solution. Unfortunately just the `typeid` operator will not solve all problems. This operator gives us the exact type of the object but this is not enough sometimes. Imagine that we have a subclass of `Student` called `FirstYearStudent`. If the pointer obtained from the list is to an object of `FirstYearStudent` then the previous code will fail because the `typeid` of the pointer and of the `Student` class will have different values.

The way around is:
List *list;
Student *s;
//...
if (s = dynamic_cast<Student>(list->get_next())) {
    int m = s->get_mark();
    //...
} else {
    // s is not a Student
    // s has the value 0 here
}

This actually solves the problem even if the object return by the list is a FirstYearStudent.

Unfortunately this extension to C++ is not yet present in all compilers. Other parts of the programs code forced us to choose a certain compiler that didn't support the RTTI extension.

During time (especially before the addition of the RTTI mechanism in C++) different other solutions have been suggested.

Inspired by Interviews [Linton92] we adopted the following solution. We added two member functions to the BasicSharedObject class:

typedef int ObjectType;
#define BASIC_SHARED_OBJECT -1

class BasicSharedObject {
    ...
    virtual int is_a(ObjectType t);
    virtual ObjectType object_type();
    ...
};

The object_type should be similar to the typeid operator of the C++. The implementation of the function is simple:

    ObjectType BasicSharedObject::object_type() {
        return BASIC_SHARED_OBJECT;
    }

The is_a function should solve the FirstYearStudent problem. The implementation of it is:

    int BasicSharedObject::is_a(ObjectType t) {
        return t==BASIC_SHARED_OBJECT;
    }

When defining a new class the two functions must be redefined. For the Person example things would look like:

#define PERSON 1234

class Person : public BasicSharedObject {
    //...
public:
    virtual int is_a(ObjectType t);
    virtual ObjectType object_type();
};

    ObjectType Person::object_type() {

return PERSON;
}
int Person::is_a(ObjectType t) {
    return (t==PERSON)||BasicSharedObject::is_a(t);
}

The last function must be carefully understand. In case of multiple inheritance the
is_a functions of all the parents must be called.

The use of the defined system would be:

    List *list;
    Person *p;
    Student *s;
    //...
    p = list->get_next();
    if (p->is_a(STUDENT)) {
        s = (PStudent*) p;
        int m = s->get_mark();
        //...
    } else {
        // p is not a Student
    }

This is a good solution for our problem. In the future we could consider changing
this to RTTI as soon as that extension is part of more C++ compilers.

Object life and Object migration

Let's see now what the life cicle of an Object is. A certain process will decide to
create an Object:

    Person* p = (Person*) NameManager::instance()->find("john");
    if (NULL) {
        p=(Person*)ReplicationManager::instance()->create(PERSON,"john");
    }

At the creation all other processes in the system will be announced about the new
Object. Each of them will create a local copy of it. The original and the copies do
not differ. No process owns the Object. All can access it, modify it or delete it. The
remote creation is detailed a bit later.

The migration of an Object appears when a certain process creates an Objects and
later ends or fails. In this case the Object will continue to live as long as one of the
processes involved in the system is still running. The location and migration of the
Objects are transparent for the user.

Remote Object Creation

One problem that had to solved was the creation of objects on the remote sites.
Assume that one site decides to create an object of class Person. Then that site
broadcasts an event saying that it did create the object. How should things happen
at each remote site?

The event will be received by the Event Manager. The event cannot be
directed to the object because the object doesn't exist yet. One solution would be the
Event Manager to create the object and name it (by this adding it in the name
Manager's table). This would be OK but if a new type of class if declared then the
Event Manager must be updated and recompiled. This is an unwanted way of doing
things. Other OO libraries use constructor functions. In there solution a user-supplied function is called when an object construction is needed. The user defines that function and passes it to the object needing it.

Our solution is closer to the OO abstraction and is much easier to use. In our solution there exists an object of each type recorded in the Name Manager and having the same name as the class. An event for the creation of an object of class Person would be directed to the object called person.constructor. The effect would be that the object (of type Person) would duplicate itself and record the new object in the Name Manager with the name supplied by the event. The code will be:

```cpp
void Person::construct(EventIn &event) {
    Person *p = new Person;
    char x[MAX_OBJECT_NAME];
    event>>x;
    p->name(x);
}
```

In the initialization of the application there must be a part where all classes are recorded in the Name Manager. This looks like:

```cpp
Person *p = new Person;
p->name("person.constructor");
Group *g = new Group;
g->name("group.constructor");
```

If an application receives a create event for an object type that is unknown to the application then the constructor object won't exist in the Name Manager so the creation if the object will be ignored. This is actually the desired behavior.

### 3.2.7 The Persistent Storage Process

In order to obtain the persistent storage of Objects we had to consider the constraints of the Ode database system. First the fact that each class must be redefined according to the O++ specifications. This redefinition (the Ode stub of each Object) can be automatically generated or can be written manual. The implementation of such a filter is out of the focus of the project. In the future such solutions could be considered.

The first major problem was the way to implement the database access. Two models have been considered: having the database access code in each CoObjects process (Figure 3.2) or having a special CoObjects process with the single mission to store the needed Objects persistent (Figure 3.3).

![Figure 3.2 - Ode code in each CoObjects process](image.png)
Chosing among the two models has been based on reliability. Some will say the first version is more reliable as there are multiple processing accessing the Ode server. Unfortunately the fault tolerance of the howl system is the one of the Ode server as all processes access it. In the case of the failure of this server or of the machine where it is running the system is dead. Recovery techniques would be difficult to implement. Additionally each process using CoObjects would have to be compiled additionally with the O++ compiler.

![Diagram of Persistent Storage Process Version]

In the second version we add to the system a CoObjects process that has the job of whatch all changes in Objects and to store the new version of them in the database. The advantage is that only this process must be compiled with the O++ compiler. Of course, the fault-tolerance is again restricted to the fault-tolerance of the Ode server but recovery techniques are very easy to implement. In case of the failure of the Ode server at the restart of it all Objects in the database will be replaced by the actual version in the PSM process (persistent storage manager process). If the PSM process fails all the Object changes are recorded in all other processes. At the restart of the PSM process (automatic) he will get the state of the system from another process and can update the database. Additionally to all this, in the case of switching to another database system only the PSM process must be changed and recompiled while in the previous model all processes would suffer changes and recompilation.

Taking all this in consideration the second model has been chosen.

An improvement of the system is a late commit version. Because writing in the database is slow we can improve the performance of the PSM process by postponing the commitment of the Object’s changes. The writing in the database will happen less frequent. If we consider the fact that some of the Objects are all the time changing, instead of writing in the database each new value we just write the last one. This will increase the speed of the PSM process as well as decrease the demand on the Ode server.

**3.2.8 Defining new Object types**

The user must create his own classes. These classes must be drived from the BasicSharedObject. In doing this the user must be aware of the following facts.

For each new class the user must redefine the `set_value()` and `get_value()` functions. These functions must marshal and unmarshal the Object data in a proper form.
The RTTI functions (is_a() and object_type()) must be rewritten and a type for the new class defined according to the previously presented method.

If the Objects of that class should be stored in the database a Ode stub for these class must be defined and the PSM process recompiled with the new definition.

3.2.9 Conference control

In order to control sessions (or conferences) we had to define some Objects and some Managers.

First the Authorization Manager. He is responsible for identifying the user of the process and check if that person is a member of the group. If not no access is granted to that person. Additional password and signature protections can be added.

Classes describing a user, a group, a machine and a process are also defined (see next chapter). These informations are kept updated by the system. In the case of a process joining the system this process will be recorded as well as the machine where the process is running and the user that owns that process.

Managers for history or logging information can be easy created based on these classes. These Managers can be used for user testing, for example.
4. Using CoObjects

4.1 @work

The first application based on CoObjects was an application called @work (read ”at work”). This application want to provide the user with part of the functionality of CoDesk.

@work has been designed to provide a sense of awareness in a group of people cooperating. The application provides information about each member of the group: if he is working, on what computer, since when, if he is available or not, when is he coming back, etc.

The application comes in three interface versions. The first one (Figure 5.1) is a graphical interface with video image. It offers a video image from all members of the group that are currently working or the picture of all others. Additional information about the station used (or used last), a message from each member to the others (like ”I will be back at 1”) and a status (away from keyboard, available, busy, private) are displayed for each person. In the lower part of the window we have a field for message passing. Each member can send messages to another person or to the howl group.

The second interface (Figure 5.2) is a Word Wide Web (WWW) filter. In the case the user is on a different computer then a Sun station (Mac, PC), he can get the same information using Mosaic or Netscape. The program will generate html information in each moment. The information is similar to the previous one (except the video that is missing). The user can receive and send messages and can set his own information.
The last interface is a text one that is meant to be used from the home computer of each member where graphics is too expansive. The information is the same one except video and photo.

As we see the information distributed is the same and the operations are similar. The only difference is the interface. Translating in the Model-View architecture, the Model is the same while the View differs. All three applications will use the same SOS module plus the same object definition while the interface will be different.

We had to define the following new classes:

```cpp
class Person : public BasicSharedObject {
    public:
        Person();
    ...
    enum PersonStat {OUT, BUSY, AFK, HERE, OTHER};
    ...
    protected:
        char* _realName;
        char* _emailBox;
        char* _gif;
        char* _telephoneNumber;
        char* _internalInfo;
        char* _publicInfo;
        ObjectName _messageBox;
        PersonStat _stat;
        EList<char> _processes; // person’s processes list
        ObjectName _machine;
```
In order to implement the message passing we defined the following classes:

class MessageBox : public BasicSharedObject {
  public:
    MessageBox();
    ...
  virtual void append_message(Message*);
  virtual void remove_message(Message*);
  virtual void remove_all_messages();
  virtual char* get_message(int);
  virtual Message* get_next_message();
  virtual int messagebox_len();
  ...
  protected:
    ELList<char> _messages;
};

class Message : public BasicSharedObject {
  public:
    Message();
    ...
    virtual void send();
    ...
  protected:
    char* _text;
The three interfaces do not contain anything special except the use of the previously defined classes. I will just give an example of such a use:

```cpp
Person* p=AuthManager::instance()->this_person();
p->set_stat((Person::PersonStat) status);
p->commit();
```
or another one:

```cpp
void list_group() {
    Group* g;
    Person* p;
    g=(Group*) NameManager::instance()->find(value);
    if (g==NULL) {
        cout<<"No group with this name found."<<endl;
        return;
    }
    if (g->object_type()!=GROUP) {
        cout<<"No group with this name found."<<endl;
        return;
    }
    cout<<"Group "<<value" has "
```

### 4.2 The new CoDesk

The next step is a new version of the CoDesk interface. This time CoObjects will be used for the object distribution and persistent storage. The interface will have additional facilities among which I would like to mention direct graphics on the windows, better awareness, etc.

The class structure will be based on the classes defined for @work. Additionally classes for rooms, applications and documents will be added.
5. Future work:

5.1 Improving CoObjects

In the future some more work will be put in the improvement of CoObjects. Main directions will be:

- implementing a partial replicated version
- finding a kind of ordering between causal and total because the causal ordering is not enough for consistency while the total ordering is too much (and too expensive) for it
  - eventually replacing SID with a new group broadcast package
  - implementing late commitment in the PSM process
  - a stream-like type of data distribution could be added
  - new implementations for different platforms could also be considered
  - implementing automatic stub and Ode stub generation

5.2 Shared Interface Server

Based on the CoObjects SOS we could implement a Shared Interface Server. Previous work done in the COMIC project could be a good start for that. Mixing CoObjects and InterViews/Fresco seems a good idea. Still we have to wait for the new Fresco release.

5.3 Other CSCW applications based on CoObjects

In the future designing and implementing CSCW applications based on CoObjects will be suggested to other researchers in the field. There experience would be a great input for further versions of the system.
References


[Lamport78a] Lamport L. - *The Implementation of Reliable Distributed Multiprocessor Systems*, Computer Networks 2

[Lamport78b] Lamport L. - *Time, Clocks and the Ordering of Events in a Distributed System*, Communications of the ACM 21 (7)


[Little93] Little C.M., Shrivastava K.S. - *Object Replication in Arjuna*


[Mullender93] Mullender S. (editor) - Distributed Systems, Addison-Wesley


[Sandor95] Sandor O., Tollmar K. - The Collaborative Desktop - An Environment for Computer Supported Cooperative Work, demonstration submission for ECSCW'95


[SunSoft91] SunSoft - Solaris SunOS 5.0 Multithread Architecture

[SunSoft94] SunSoft - pthreads and Solaris threads

[Sun85] Sun Microsystems - Remote Procedure Call Protocol Specification

[Stroustrup91] Stroustrup B. - The C++ Programming Language, second edition, Addison-Wesley

[Stroustrup92] Stroustrup B., Lenkov D. - Run-Time Type Information for C++, C++ Report, March

[Stroustrup94] Stroustrup B. - The Design and Evolution of C++, Addison-Wesley

[Vinoski93] Vinoski S. - Distributed Object Computing with CORBA, C++ Report, July-August