A FRAMEWORK FOR MIGRATING OBJECTS IN DISTRIBUTED GRAPHICS APPLICATIONS

by

Vijay Machiraju

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

The University of Utah

June 1997

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Vijay Machiraju

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Gary Lindstrom

Richard F. Riesenfeld

Peter Shirley

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Vijay Machiraju in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Gary Lindstrom
Chair, Supervisory Committee

Approved for the Major Department

Thomas C. Henderson
Chair/Dean

Approved for the Graduate Council

Ann W. Hart
Dean of The Graduate School

ABSTRACT

The complexity involved in developing distributed applications has decreased with recent developments in distributed object-oriented platforms and emerging standards such as the Common Object Request Broker Architecture (CORBA). Graphics applications including three dimensional graphics packages, CAD/CAM applications, and user interface management systems are one class of applications that benefit from these recent advances. However, these applications have some special requirements and constraints such as the existence of many fine-grain objects and object graphs, the need for fine-grain sharing of objects, and collaboration. CORBA does not support fine-grain objects and their sharing by different clients. We show how graphics applications can benefit from an underlying support for fine-grain objects and large-grain objects. We also show that *object migration*, and migration of objects of all granularities in particular, provides an elegant solution for solving the problems of distributed graphics applications mentioned above. We propose and implement a CORBA-based architecture that supports *copying, replication,* and *complete movement* of object graphs and objects of *all* granularities. Other issues such as concurrency control, which arise from sharing or replicating objects and object graphs, are also addressed from the stand point of graphics applications. All these ideas have been tested on sample applications in the Alpha_1 geometric design and modeling system.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

With the increase in the popularity of the internet and with the change in computer systems from the *time-shared model* to the *network-as-computer model*, distributed systems have attracted the attention of many researchers. The semantic gap between the functionality offered by a conventional operating system and the abstractions required by a programmer using a higher-level language is aggravated by distributed systems. An operating system typically offers a low-level interface to the network communication primitives which makes the development of distributed applications difficult. This motivated the development of many *distributed platforms* such as OSF DCE, OLE, and CORBA. Distributed platforms simplify the process of developing distributed applications by shielding the developer from the aspects of distribution. Though there have been many advances in distributed platforms, applications such as CAD/CAM systems and other graphics packages cannot directly benifit from these advances for several reasons. This chapter first describes some of the issues involved in developing distributed applications and explains how *object-oriented programming* and *distributed object-oriented platforms* have helped in reducing the complexity involved in designing and developing a distributed application. Then, it focuses on the issues in graphics applications that are not particularly supported by distributed systems and platforms. The rest of this thesis aims at solving the problems associated with developing distributed graphics applications.

1.1 Distributed Applications

Tanenbaum defines a distributed system [1] as a collection of independent computers that appears to the users of the system as a single computer. Distributed applications are special-purpose distributed systems which should provide the users with the same "virtual uniprocessor" view. Distributed applications have many advantages such as resource sharing, fault tolerance, concurrency, collaboration, and economics.

Resource Sharing: Many applications are inherently distributed. For instance, a bank, which has many branches geographically distributed across the world, needs to share all the information about its customers and other assets, so that the customers can use the services offered by any of the branches. Similarly, resources such as printers and scanners could be shared by all the computers on a network. It is highly unlikely that every computer would be connected to its own printer or scanner. In essence, resource sharing can be either *data sharing* or *device sharing*.

Fault Tolerance: A task in a distributed system is spread across several nodes. Hence, the failure of any node in the system will only affect a portion of the task. Ideally, failure of 10% of the nodes should result in a performance degradation of only 10%. Distributed systems allow applications to move from one node to another, or to replicate certain important computations on multiple nodes. This increases the reliability of systems despite intermittent network failures.

Concurrency: Distributed systems allow more efficient use of the resources on the network, through load balancing. This increases the concurrency between the applications, and between various parts of the application, which results in increased *throughput* of the system.

Collaboration: Computer Supported Collaborative Work (CSCW) is one of the areas of growing interest among researchers and software designers. This would allow designers from multiple backgrounds and distant locations to work together on an application. For instance, a human interface expert and user interface programmer, from two distinct locations, could develop an ideal user interface through CSCW. Such collaboration could be made possible only through distributed systems.

Scalability and Incremental Growth: Requirements in an organization change with time. When the expectations increase, it would be easier to add more systems to the existing system, than to replace the entire system. Distributed systems are easily scalable, since they allow new systems to be added to the existing ones, incrementally.

Economics: Perhaps the greatest motivation in using a distributed system comes from economic factors. It would be difficult to increase the throughput of a single system beyond a certain limit. Further improvements in performance can be obtained only by connecting multiple computers together. Thus, a distributed system offers a better *cost-to-performance* ratio than a mainframe system.

Since a distributed application should shield the users from all the details of networking and distribution, developing such an application becomes a nontrivial task, without any support from convenient programming paradigms and underlying operating systems. Object-oriented design and development has provided a convenient programming paradigm, and the recent developments in distributed platforms have solved the problem of providing underlying support.

1.1.1 Object-oriented Design

An *object* is a self-contained unit of software functionality containing both data and procedures for working with that data. Under the object-oriented paradigm of programming, programs are composed of objects that encapsulate certain specified functionality. Such a design provides the developer with many advantages, such as component integration through clearly defined interfaces, ease of design through abstraction, and reuse through inheritance. These advantages are discussed below in detail.

Component Integration: Components are entities that are not bound to any particular program, language, or operating system. Well-planned and designed objects serve as components in that they can be reused in many applications. Objects are identified by well-defined interfaces. This notion of objects results in the fact that several objects, which are built independently, can be put together in a meaningful way in an object-oriented environment. For example, a matrix object (which provides several matrix operations) and other geometric objects can be put together to create a rendering application in graphics. The same matrix object could be used in a mathematics tool to solve a set of linear equations. Component Integration is one of the main motivations in the development of many object-oriented distributed platforms such as OLE and CORBA.

Abstraction, Information Hiding, and Encapsulation: *Abstraction* is a way of specifying what information is important and what is not, in the context of a particular problem. There are different kinds of abstractions, such as functional abstraction and data abstraction, that give importance to different kinds of information. *Information hiding* specifies the visibility of the information that is classified as less important by an abstraction. For instance, in functional abstraction, implementation of the functions is hidden, while making their interfaces visible. *Encapsulation* is covering a collection of items with a well defined boundary. The boundary can be transparent, translucent, or opaque, depending on the extent of the information that is to be hidden from the user. It should be noted that, encapsulation can exist without any information hiding. For instance, the `struct` definitions in C language provide for a method of encapsulation, where all the information inside the structure is visible to the user. On the other hand, classes in C++, provide a method of encapsulation in which certain portions of the class are visible or transparent (public members), certain portions are only partially visible or translucent (protected members), and certain portions are invisible or opaque (private members).

An object-oriented language provides these three features, which allow an object to be specified in terms of a well-defined interface. The interface serves as a contract for all the users, independent of the implementation and the language used for the implementation of the object. Using these features also enables *reuse* of objects and *portability*.

Inheritance and Dynamic Binding: *Inheritance* is a mechanism provided by most of the object-oriented languages, where a type (called a *subclass* or *derived type*) can inherit all the characteristics of another type(s) (called a *superclass* or *base type*). Inheritance is used for two purposes - *specialization,* in which a derived class can change the implementation of the methods provided in the base class without changing the interfaces of the base class, and *extension,* in which a derived class can add more data members and methods to those of the base class. C++ provides for both *interface inheritance,* which allows the type of a derived class to be inherited from that of its base class, and *implementation inheritance,* whereby the implementation of the methods in the base class can be reused in the derived class. Inheritance enhances the *reuse* and *quality* of software. Programmers can reuse the types and implementations in the base class instead of writing new code. They can use the well-tested base class code in the derived classes, and this helps in reducing programming errors. Use of inheritance also results in decrease in the size of the compiled code for the reasons mentioned above.

In *dynamic binding*, the type of an object (and thereby, its associated operations and their implementations) need not be known fully until run time. Dynamic binding is available in C++ via the use of the keyword `virtual`. Though, there is some run-time overhead involved in the use of dynamic binding, the advantages that it provides from the standpoint of flexibility and extensibility of software, far outweigh its disadvantages, in certain applications. *Flexibility* is the ability to easily recombine existing components into new configurations, and *extensibility* allows easy addition of new components. Thus, inheritance and dynamic binding support evolutionary, incremental development of reusable components by specializing or extending a general interface or implementation.

Polymorphism: Polymorphism allows an application to make a request on an object without knowing the exact type of the object. Objects in different classes receive the same message, yet react in different ways and provide the appropriate behavior. Polymorphism lets an application view two different objects through a common interface, eliminating the need to distinguish between related objects. Overloading is a variant of polymorphism by which one can define different versions of the same method, discriminated by different parameters.

1.1.2 Distributed Objects

Classical objects (in the C++ sense) are known only in the program that creates them. On the other hand, *distributed objects* can reside anywhere on the network. Remote *clients* can access them through method invocations. Actually, programming in an object-oriented way makes more sense in a distributed setting. In a single-user, single-machine setting, programmers may be reluctant to use all the features of object-oriented languages for efficiency considerations. For example, they might resort to global variables and monolithic programs to avoid passing parameters and calling functions. Such a style would not be possible in a distributed application, since we still lack efficient implementations of distributed shared memory. Further, as we will see in the next section, developing a distributed application adds a whole set of new requirements, which makes these applications harder to develop and maintain.

1.1.3 Distributed Platforms

Despite the many advantages mentioned above, distributed systems have some fundamental problems, such as latency, network failures, and security. Hence, developing a fail-safe distributed application, whose components collaborate efficiently, transparently, and scalably, is difficult. There are several differences in the invocation of a local object as opposed to calling a method on a remote object, which makes distributed application development harder. To give a clear idea of the level of details that are to be handled, consider the case of a system that provides a socket interface to an application developer. In addition to developing the normal application code, the developer has to incorporate the following features into his application:

Location of the Object: The program has to keep track of the network address of the service provider.

Connections: The program has to establish the network connection to the service, and close the connection after all the services are completed.

Marshaling: The server object may have been compiled on a different architecture and/or using a different compiler compared to that of the client. This would mean that the server might have a different representation of the data types, different alignment for fields within a structure, and different conventions for parameter passing. Hence, the client has to *marshal* its parameters in a form that is suitable for the server.

Type Checking: Since socket descriptors are weakly typed, it would be difficult to check for the types of the parameters at compile time.

Error Handling: In a network environment, errors can occur during the transmission of messages. So, the client has to handle these conditions, and appropriately retransmit the messages.

Security: The client application code has to handle all the security considerations that would be involved in transmitting a message across a network.

Portability: Using a particular interface, such as sockets, would limit the portability of an application. Major parts of the client code have to rewritten when this interface changes to something else, for instance, an Inter Process Communication (IPC) object library.

Other major issues that are involved in designing a distributed object system (or a distributed object platform) are:

Object Location: Some systems fix the positions of the objects once they are created, whereas others allow migration of objects. Object migration is considered at length in the next chapter, and in the reminder of this thesis.

Transparency: There are several features in a distributed system that can be kept transparent to the user. For instance, the location of objects could be handled by the system, or could be left to the user. There are several disadvantages, as mentioned above, in leaving such details to the application developer. Further, if a system allows for moving objects, the user can be informed about the migration, or the system can keep track of the new location, and forward all the invocations to the new location. Developing a fully transparent system would entail significant overhead and inefficiency in a distributed system, whereas a system which gives full control to the users would shift the difficulties to the application developer.

Processor allocation: There are two main models by which processes in a distributed system are allocated to processors - the *workstation model* in which each user executes processes on exactly one machine, and the *processor pool model,* in which all users have equal access to all processes. Most of the distributed systems fall somewhere between these two extremes.

Communications: Messages between objects can be exchanged in many ways - synchronous, semisynchronous, or asynchronous. In *synchronous* mode, an object (or an application) that has invoked a method on another object blocks till it gets the response. In *asynchronous* mode, the client that has invoked the operation will be interrupted when the response is available. So, the client will not be blocked for the response. In a *semisynchronous* mode of operation, the client polls for the response periodically while continuing its normal operation.

Multithreading: Multithreading an application would result in increase in performance, at the cost of portability and increase in development and debug time.

Server Concurrency: Servers in a distributed environment can be of two types: iterative servers and concurrent servers. *Iterative servers* queue up requests and handle them in FIFO fashion. *Concurrent servers* use multithreading to simultaneously handle requests from multiple clients. Iterative design is most suitable for short duration services that exhibit little variation in execution time (e.g., unix commands such as `time` and `echo`). On the other hand, concurrent servers require more stringent scheduling strategies and synchronization mechanisms, and are more suitable in internet services such as `ftp`. Changing from one implementation to the other should not affect the application clients.

Deferred Activation: Objects can be activated in the servers when a method is invoked on them. In the rest of the time, they can be in a *dormant* state. When a method is invoked on a dormant object, it can be reactivated. Deferred activation would be appropriate for less frequently used objects, and when the system is composed of a large number of objects. Such a mechanism would also reduce system load, and increase the efficiency in the use of available resources. Deferred activation, if implemented, should be completely transparent to the user.

Persistence and Object Lifetime: An object is said to be *persistent* if it can live outside the process that has created it. Object persistence could be achieved by storing the state of the objects on a persistent storage device. However, there are several issues that makes the implementation of persistence harder, such as references to other objects within an object, and transparent translation of the object state between memory and disk. Persistence, coupled with checkpointing, would be useful in a distributed application since clients should be able to access server objects inspite of network failures.

Heterogeneity: A distributed system will often be composed of machines with different architectures, and running different operating systems. Objects in a distributed system should be able to interact with other objects in the system, irrespective of the machine and operating system on which they are being executed. Heterogeneity in architectures leads to differences in parameter passing techniques, byte ordering differences (little-endian or big-endian), and conversions between binary formats supported by the architectures. Operating systems differ in their support for multithreading, shared memory implementations, and system call interfaces (e.g., POSIX or Win32).

Network Protocols: Building a system that supports many LAN and WAN protocols would be difficult to implement. TCP/IP, X.25, ISO OSI, Novell IPX / SPX are some of the common protocols that handle certain communication features in very different ways.

Consistency: Several problems in consistency arise due to caching on clients and servers. On the other hand, caching is essential for reducing communication overheads. Problems in consistency also arise when messages are in transit, or when objects are replicated.

Garbage Collection: Garbage collection can be very complex in a distributed environment, since a process on another machine can be holding the reference to a particular object. Garbage collection should be *safe* and *lively*. Objects with references should not be garbage collected (safety) and objects with no references should be garbage collected (liveness).

A *Distributed Object Platform* is a utility that isolates some or all of these concerns from the distributed application developer. It provides a high-level interface, so that developers can focus on the specific application requirements. Examples of such platforms include the *Common Object Request Broker Architecture (CORBA)* and *Object Linking and Embedding (OLE)*. We take an in-depth look at CORBA in the next chapter.

1.2 Special Issues in Graphics Applications

Graphics applications such as 2D and 3D graphics packages, CAD/CAM applications, and user interface management systems (UIMSs) are one class of applications that benefit largely from a distributed object-oriented design. The *Alpha_1* geometric design, modeling, and manufacturing system at the University of Utah is one such application that is composed of many components such as a model graph constructor, a model-viewer, and a renderer, in addition to all the graphics objects such as points, curves, and polygons. These components can exist anywhere on the network, and each of these components needs to interact with the others. In addition to the issues raised by common applications, as discussed in the previous section, graphics applications raise an additional set of issues. These issues, which are discussed below, are the main motivation for the work in this thesis.

Resources: As discussed earlier, one of the main advantages in using a distributed system is resource sharing. This can be easily observed in graphics applications. Graphics applications require special capabilities such as graphics workstations (for user interfaces, animation, and rendering), computational workstations for executing compute-intense algorithms (such as rendering algorithm or transformations on graphs of objects), color printers, and scanners. Some of the architectures have special built-in capabilities, such as z-buffers, for displaying graphics, while some of them have high computational capabilities. Better utilization of these resources would be possible if the applications were distributed.

User Interfaces: One of the chief requirements in the design of user interfaces is their *isolation* from the underlying application. The abstractions used in building user interfaces (such as menus and buttons) should be distinct from those used in building the application (such as polygons and curves). There can be communication between these abstract objects, but each of them should not depend on specific implementation details of the other. In an isolated system, it would be easy to change certain parts of the implementation, or certain parts of the interface, without affecting the whole system. For example, it should be possible to change the interfaces from command-line interaction to menu interaction without any change in the application. With such isolation, applications can be running on one system, while the interface can be presented to the user on another system, e.g., one with powerful graphics capabilities. Another aspect that should be considered in designing good user interfaces is that they should be able to put together components from different parts of the system into a single window. Fresco [2] is an example of such interface development tool, which would help in distributing components of the interface across a collection of machines. An interface built with such a tool can, for instance, take a picture from a CAD database, a message from another location, compose them with some local interface options, and form a user interface in a single window.

Interaction and Response Times: Interaction is one of the key aspects of a graphics application. Most of the graphics applications provide some method of interaction with the objects involved. *Immersive environments* and *virtual reality* take the interaction a step further, involving the user completely in the interaction. Response time, which is the time lag between the user input and the response of the system, is one of the main considerations in designing good applications and interfaces. In particular, virtual reality applications impose real-time requirements on the interaction. As network delays are involved in a distributed application, a distributed graphics application designer should give high importance to interaction, and consider ways of reducing response time.

Granularity of Objects: A graphics application typically is composed of objects of varying *granularities*. *Large-grain objects* are characterized by their large size, relatively large number of instructions that they execute to perform an invocation, and relatively few interactions they have with other objects. They often reside in their own address space, and are thus *heavy weight* in nature. Examples of large-grain objects in a graphics application are a model-viewer, a model-constructor, or a renderer. On the other hand, *fine-grain objects* are characterized by their relatively small size and large number of interactions with other objects. They share the address space of the process in which they are created, and there can be many fine-grain objects within one address space. Example of fine-grain objects in graphics applications are the various graphics objects such as points, lines, polygons, and curves. Thus, a graphics application has a few large-grain objects and a large number of (typically, $10,000$ in one large model) fine-grain objects. The platform we choose for such an application should provide support for objects of all granularities, by providing efficient interaction mechanisms between fine-grain objects, and uniform interfaces for all objects to the application developer.

Object Graphs: Most of the graphics applications build graphs of objects. The Alpha_1 system uses a *model graph* to represent the dependencies between objects. For instance, a *line object* can be dependent on two *point objects* that define the end points of the line. Similarly, the line can be a part of an *outline curve object*. These dependencies can be modeled using a graph as shown in Figure 1.1. Such a graph can be used to propagate the changes made in any of the objects involved, to all its dependents, recursively. For example, when the coordinates of the point A in the graph are changed, the line, and hence the outline could be updated. Object graphs introduce dependencies between objects in the form of *containment*, where one object is contained within another object, or *reference*, where one object contains a reference to another. Object graphs introduce special problems when objects are migrated,

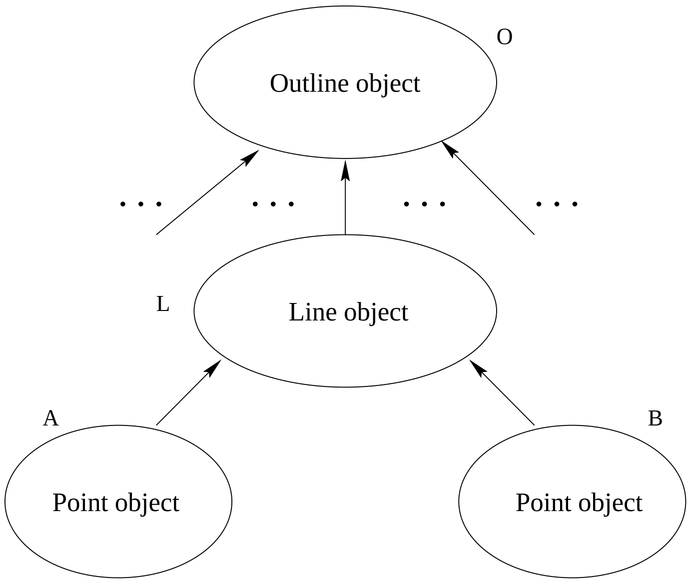

Figure 1.1. An example of an object graph

or for implementing object persistence. There are also several issues that need to be addressed while deciding the locations of objects within a graph, since they need to interact frequently.

Fine-grain Sharing: Dependencies in a graph propagate as seen in the above example. This restricts graphs from being replicated on multiple nodes. However, it can be observed that multiple users can alter independent portions of the graph. For instance, we can let object A in the graph of Figure 1.1 to reside on node X, and all other objects on node Y. This introduces the notion of *fine-grain migration* of objects within a graph, and exerting control restrictions on these objects. There are no easy mechanisms to decide which group of objects in a graph should be migrated. These decisions are, in general, dependent on the application semantics.

Collaboration: Collaboration in design and development is very common in graphics applications. Not only would users like to work on the same design, but they would also want changes made by one user to be reflected immediately on all the machines, in the ideal case. Collaborative design has been successfully demonstrated in the Alpha_1 system by running the `c-shape-edit` server (which is a model-graph constructor) on one machine, and having the model-viewer clients on different machines. However, there are several deficiencies in this collaboration. The manipulated object is local to one of the users. Further, there are no mechanisms that would enable one user to control the views of the other users. Efficient interaction with the underlying model, by all the users, would require replication of the object graph on all the clients, so that the users can manipulate independent portions of the graph. This would also bring in consistency requirements between graphs of replicated objects, which is a nontrivial task.

Existing distributed platforms such as CORBA do not support fine-grain objects. They also do not provide any special purpose solutions to application specific problems such as the ones mentioned above. In this thesis, we explore solutions to these issues and show that *migration of objects of all granularities* is an effective means to:

- utilize resources on the network,
- support interaction between objects in an object graph,
- provide fine-grain sharing and interaction of objects,
- reduce the response time in interactive tasks, and
- implement a fault-tolerant application.

We propose and implement object migration on the top of a well-standardized distributed platform, CORBA, which resolves most of the issues raised in section 1.2 effectively. We use an implementation of CORBA, called *Orbeline* from PostModern Computing. We consider and implement several migration policies such as *replication, copying,* and *complete movement.* Migration of large-grain and fine-grain objects, granularity issues involved in migration of object graphs, fine-grain sharing of objects in a graph, and concurrency control on shared objects, are also considered. Finally, all the implementations are tested on a distributed object-oriented implementation of Alpha_1.

The rest of this thesis will be organized as follows. Chapter 2 defines object migration and discusses the issues involved in migration in greater detail. Other related works and how our model compares with the existing implementations of CORBA, are also discussed in Chapter 2. Chapter 3 discusses *object services* which provide the framework for implementing object migration. In Chapter 4, we explain how object services can be used to implement migration of large-grain and fine-grain objects. The complexities involved in migration of object graphs are considered in Chapter 5, and concurrency control strategies for maintaining consistency between replicated objects are elaborated in Chapter 6. All our ideas will be tested on a few sample applications of Alpha_1, and the results will be presented in Chapter 7. We summarize our conclusions in Chapter 8.

CHAPTER 2

OBJECT MIGRATION

A distributed object-oriented system is composed of many objects that can reside anywhere on the network. These objects interact by exchanging messages with each other. *Object migration* involves moving an object from one node to another on the network. The original node on which the object was located is called its *source*, and the final node is called its *destination*. The migrated object, then, becomes local to the destination, and the destination can invoke methods on the object locally, without resorting to any remote invocations.

There are many reasons why one would like to migrate objects. Some of the most common reasons are summarized below.

Load Balancing: Migrating objects reduces the workload on the source machine. Load leveling across the machines in a network can be obtained by migrating objects, as workloads change dynamically. This increases the overall performance of the system.

Resource Sharing: Resource sharing was cited as one of the primary motives in using a distributed system. Object migration provides for an improved way to share resources. Objects that need particular resources that are located remotely can be moved to the corresponding nodes.

Fault Tolerance: Objects can be copied or replicated on multiple nodes. When one of the nodes containing an object fails, the system can still continue its normal operation using the objects located on other nodes. Further, objects can be migrated to other nodes, during *graceful degradation* of a faulty node.

Less Communication Overhead: A local method invocation involves less overhead than a remote invocation. A remote invocation inevitably involves parameter marshaling delays on both sides and network delays. By bringing the required objects to the working node, all the remote calls can be converted into local calls. This decreases the response time in interaction, which was cited as one of the requirements for a graphics application.

Granularity in Object Graphs: By replicating parts of the object graph on different nodes, users can update the graph simultaneously. Objects can also be replicated with different access rights. Thus, an object that is being modified by one user, can be viewed by a different user. This increases the concurrency, and hence the throughput of the system.

System Administration: By clustering related objects together, kernel operations such as I/O and page faults can be amortized. For instance, if two objects require access to the same data structures, the number of total page faults can be reduced by *co-locating* the two objects, and using the data structures that are already in memory.

Persistence: In general, persistence is implemented as migration of objects to and from a stable storage. Hence, object migration simplifies the implementation of object persistence.

There are more advantages that come about from movement of fine-grain objects. Some of these are listed below.

Data Movement: Fine-grain object migration provides for an efficient mechanism to move light-weight objects and other small data items between nodes. In the absence of such a scheme, we need to make use of inefficient coarse-grain techniques such as file transfer.

Invocation Performance: The parameter objects can be moved to the location of the remote server object before invoking any methods on it. This would provide for the conventional *call by value* semantics of programming languages.

Garbage Collection: Fine-grain object migration simplifies distributed garbage collection, since all the objects can be moved to the sites where their references exist.

We observe that object migration solves a number of issues that were raised by graphics applications, in Chapter 1. Object migration does not come free of cost, and there are certain disadvantages associated with migrating objects.

Migration Overhead: Migration can result in costly time overhead, if the migrated objects are large, and if there are very few methods invoked on the migrated object at the destination. Object migration will be useful only when the estimated overhead in remote invocations on the object are greater than the overhead involved in migrating the object itself. Another case where object migration can result in poorer performance is when two clients try to migrate an object simultaneously. To overcome these problems, there should be some control mechanisms on object migration.

Security: There should be more stringent security restrictions when objects can migrate on the network. This was one of the reasons for the unpopularity of object migration in the past. With more and more research in security, and with the development of languages such as Java, which allow the entire code to be migrated, people are willing to allow objects to be moved to their machines.

Hard to Implement: There are several possibilities to be considered, and several decisions to be made, before implementing a scheme for object migration. Some of these possibilities are discussed in the next section. In the absence of portable and standard distributed platforms, there were some attempts to provide operating system support for object migration. This has not come out to be very successful, in terms of giving the user, control on the granularity of migration.

2.1 Issues in Object Migration

There are several options along several dimensions that are available for implementing object migration. In some cases, implementors have to decide on an appropriate option, whereas in other cases, they can provide multiple options to the users.

Degree of Migration: Object migration is an imprecise word which can be made more specific by the degree of migration. *Copying* an object involves making a copy of the object at the destination. The original object (which we call the *target*) and the copied object do not share any resources in common; the copied object gets a new *identity*. *Replication* is copying an object, but in this case, the target and the replicated object share the same identity. Updates to a replicated object should be reflected in the target, and vice-versa. Replication requires concurrency control strategies on the object involved. An object can also be *completely migrated* from the source to the destination. In this case, the target object ceases to exist on the source, and will be identified with the same identity on the destination.

Object Kind: There are different kinds of objects in a distributed object system. Objects can be classified as *passive* or *active* depending on whether they are associated with any server process or not. They can be classified as *quiescent* or *nonquiescent*, based on the stability of their state. If there are any methods that are active on the object, then it is nonquiescent. More fundamentally, objects can be *stateful* or *stateless (or immutable)* depending on whether they have any data members encapsulating the state of the object.

Migrating an active object involves migrating the associated process along with the object. This introduces all the concepts of process migration, which in the limiting case includes migrating the user space of the process (including text, data, and stack segments), and migrating the kernel state (including the information stored by the kernel for switching contexts, register contents, and condition codes, resources such as open file handlers and message channels, and the entire environment including the process id, user id, current working directory, signal masks and handlers, resource usage statistics, timers, and references to parent and child processes).

To migrate nonquiescent objects, either the system has to wait for the current processes to finish execution, or it should migrate the the currently executing processes along with the object. To migrate stateful objects, we need to define operations to externalize the state of the objects. This can be further complicated if the state of the object involves system resources such as file descriptors, which may not be identical across the entire system (for instance, if there is no underlying common network file system).

Server Policies: For active objects, the server that is associated with the object can have different *server policies*. In a *shared server policy*, multiple active objects share the same server, whereas only one object can be active at a given time in a server in an *unshared server policy*. In a *server-per-method policy*, each invocation of a method is implemented by a separate server being started. Object migration is dependent on the policy being used. For example, in a shared server policy, migrating one of the shared objects will be complex, if these objects share some resources, or interact with each other.

Related Objects and Object Graphs: Objects often *contain* or *reference* other objects. For example, Figure 2.1 shows an object A referencing another object B. When the object A is migrated, the reference to object B is no longer valid. We either need to migrate B to the new location, and reestablish the relationship between the objects (as in 2.1c), or need to convert all the invocations on B to remote invocations.

Heterogeneous Architectures: Migrating an object across heterogeneous architectures adds more challenges to the implementation. This is generally achieved with the help of *meta-data* which are used to convert data from one architecture to another. For example, if we need to migrate the state of an

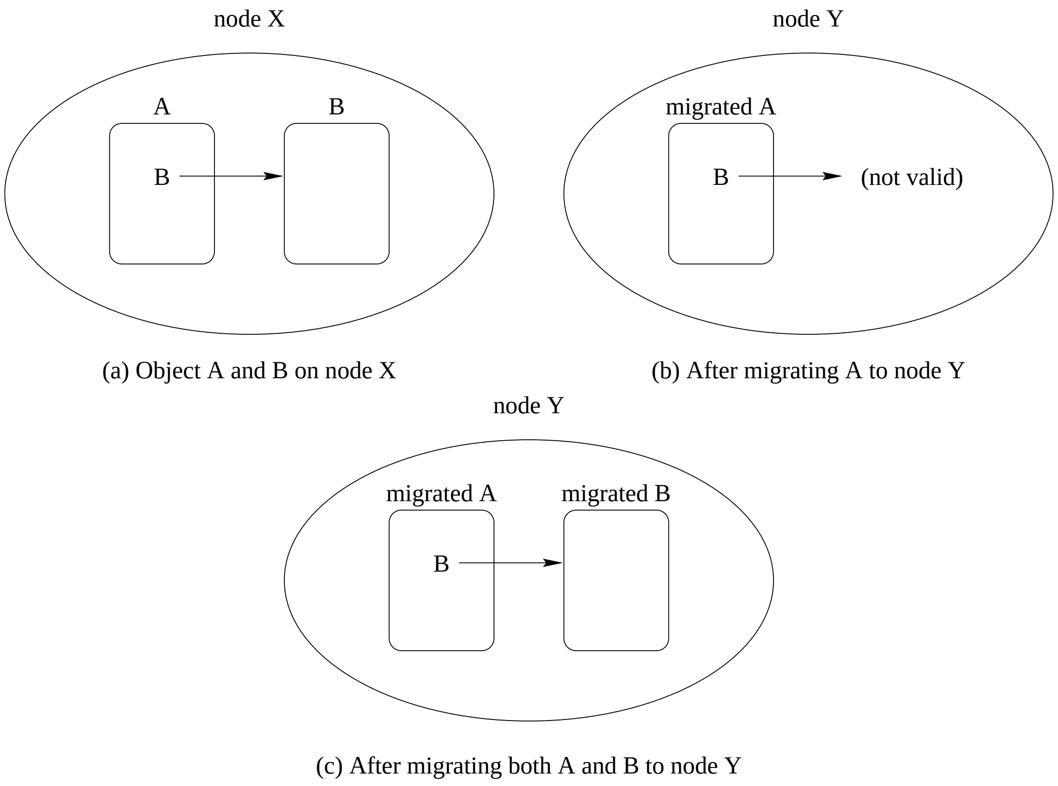

Figure 2.1. Effect of migration on related objects

object across heterogeneous architectures, it will be essential to ensure that the definition of the corresponding class of the object exists on the destination architecture. Otherwise, object migration will involve *class migration*, and recompiling the class on the destination.

Object Location: The implementation can opt to hide the details about the new location of the migrated object from the client applications, or decide to inform the clients about the new location. In the former case, the implementation has to handle all the messages that arrive at the old location, appropriately, by forwarding them to the new location. The burden on the application programmer will increase if the client has to keep track of the object location.

Residual Dependency: The on-going need for a host to maintain data structures or provide functionality for an object even after the object migrates away from the host, is known as *residual dependency*. Residual dependency is undesirable, since it affects the reliability and performance of the system while increasing its complexity. The state of an object is now distributed across many machines, and this is undesirable for the reliability of the system. Further, forwarding every message to the new location through the old location defeats the purpose of object migration, which is intended to reduce the network delays.

Implementing Transparency: It was pointed out that transparency to the user is one of the main concerns in implementing migration. Implementing transparency is difficult for several reasons. For complete transparency, the user, and every other system (including devices and files), should be completely unaware of the migration. For instance, the processes that are associated with the migrated objects should still exist in the process table of the original host. Users should be able to kill, stop, or restart these processes, just like any other local process. In general, complete transparency is difficult to achieve. This leads to different levels of transparency, such as

1. *Location transparency:* The location of the object need not be known to the clients before and after migration.

2. *Access transparency:* The invocation of the object is the same wherever the object is located.

3. *Migration transparency:* There is no noticeable difference in latency, in method invocations, before, during, or after migration.

4. *Fault transparency:* Objects migrate before node failures so that the user is not aware of the failures.

Communication Channels: All the communication channels with the clients need to be closed before migration and reopened after migration. All the messages that are received during migration should be handled. This could be done by leaving a proxy object at the original host, and requiring it to forward all the messages to the migrated object after the communication channels are reestablished. An easier solution would be to refuse all the messages during migration, requiring the client to retransmit the requests later.

Auditing and Statistics: The system should have some kind of log to record the usage statistics of objects by different clients. This can be used in certain decisions regarding the destinations for migrating objects.

2.2 CORBA

Object Management Group (OMG), which is a consortium of several companies including Sun, HP, DEC, and IBM, attempts to define the various facilities that are necessary for distributed object-oriented computing [3]. This definition, which is popularly known as the the *Object Management Architecture (OMA) Reference Model*, is shown in Figure 2.2. The central component of this model is the *Object Request Broker (ORB)*, which is responsible for transparent communication between objects. In other words, it enables objects to make and receive requests and responses in a distributed environment. *Object Services* are a collection of basic services (interfaces and objects) that provide basic functions for using and

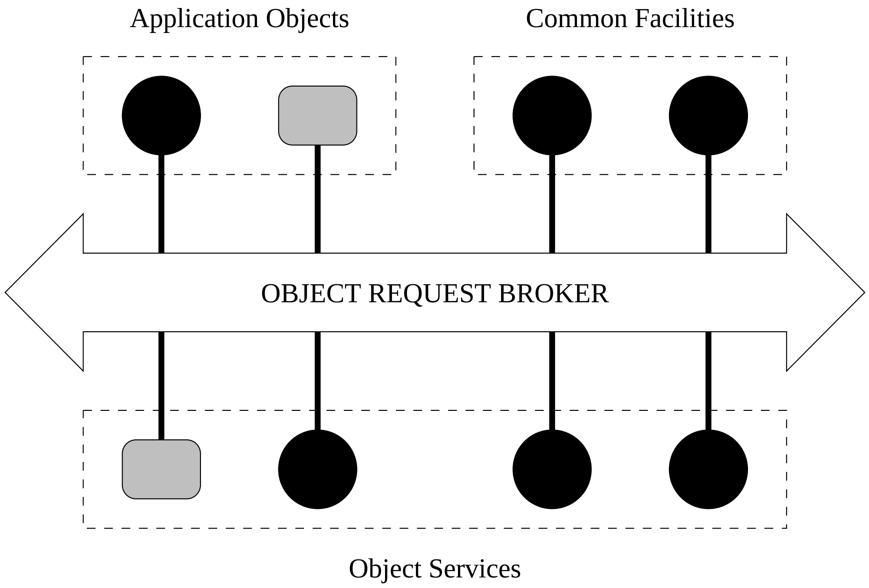

Figure 2.2. OMA reference model

implementing other objects. We shall look at many examples of Object Services, and how they are implemented, in the next chapter. *Common Facilities* are a collection of services that provide general purpose capabilities, such as email, which are useful in many applications. *Application Objects* are objects that are part of a specific distributed application. OMG defines standards for the ORB, the Object Services, and the Common Facilities.

The *Common Object Request Broker Architecture (CORBA)* [4] is an OMG defined standard for ORB in the OMA reference model. CORBA defines all the interfaces that are available through or by the ORB to the objects. It addresses the problem of making distributed application development no more difficult than developing a centralized application. Thus, it isolates the application developer from most of the issues discussed in Section 1.1.3, and provides an infrastructure for integrating application components into a distributed application. For instance, CORBA:

1. isolates the details about object location from the client programs. It keeps track of the information regarding the various servers running at different locations on the network, and provides for a transparent mechanism for the

clients to invoke methods on the server objects.

2. transparently establishes and breaks communication channels when the clients invoke methods on the remote objects.

3. marshals the parameters before transmitting them to and from the server objects. The clients need not consider the architectural and compilation differences in data formats that exist between different machines.

4. performs static compile time checking (using *stubs*) and run-time checking (using *interface repositories*) to avoid unexpected failures in the system.

5. provides for several policies, such as *atmost once* or *best effort* transmission schemes, to support retransmission during network failures.

6. supports different architectures, operating systems, and network protocols.

7. allows servers to be multithreaded or concurrent. Further, these servers can be changed from iterative servers to concurrent servers, and vice-versa, without affecting client applications.

It should be noted that inherent problems with distributed systems, such as latency, network failures, and security, still exist in CORBA. CORBA implementors should take necessary steps to handle these problems.

Objects use the *Interface Definition Language (IDL)* to define their interfaces. IDL is a C++-like language which supports interface inheritance. Clients get to know about the services offered by objects from these interfaces. As an example, the interface of a simple math object is shown in Figure 2.3. The math object supports three functions - `seta()`, `setb()` set the state of the math object, and `add()` returns the result of adding the two numbers from the state.

Compiling an IDL specification would result in stubs and skeletons. A *stub* is a proxy class that can be compiled with the client application. For the example above, the stub consists of a class called `math`, whose functions transparently invoke the remote math object, get the results, and deliver them to the client. The client

```
interface math {
     void seta(in double a);
     void setb(in double b);
     double add();
};
```

Figure 2.3: IDL interface for math object

will be completely unaware of the remote invocation. A *skeleton* is a class that can be compiled with the server application. The skeleton will be responsible for generating upcalls on the server object when any of the clients invoke a method on that object.

A typical client application, which connects to a math object and performs some invocations, is shown in Figure 2.4. The client performs some initialization, and connects to a math object. After that, the client can use the reference for invoking methods, just as in C++.

The server, on the other hand, needs to implement the functions that are provided by the math object and instantiate a math object. The code in Figure 2.5 shows how the server implements a math object, does some initialization, and waits for requests. Note that the server interacts with the *Basic Object Adaptor (BOA)* to register the implementation of the object, and to receive requests. BOA is a CORBA interface available for the server objects, to obtain object references and register server implementations. BOA is also responsible for keeping track of which object references correspond to which objects, and thus helps the ORB in locating the required objects.

In this example, we have seen several components of CORBA, such as the ORB core, IDL stubs, IDL skeletons, and the BOA. In addition, there are other interfaces to CORBA. The *Dynamic Invocation Interface (DII)* offers a convenient mechanism to form requests on object servers at run-time, as opposed to the IDL stubs, in which all the calls have to be defined at compile-time. The *Dynamic Skeleton Interface (DSI)* is the counterpart of DII on the server side. All the components of CORBA are depicted in Figure 2.6.

```
main(int argc, char **argv) {

    // initialize the ORB
    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

    // bind to a remote math object
    math *m = math::_bind();

    // use the math object to invoke methods on it
    m->seta(3.1);
    m->setb(4.3);
    cout << m->add();

    // release the math object reference
    CORBA::release(m);

    return 1;

}
```

Figure 2.4: Client application for math object

```
// implementation of the math object
class _im_math : public _sk_math {
    CORBA::Double x, y;

public:
    // constructor
    _im_math() : _sk_math(NULL) {};

    // implementation for methods
    void seta(CORBA::Double a) { x = a; }
    void setb(CORBA::Double b) { y = b; }
    CORBA::Double add()        { return (x + y); }
};

// server's main
main(int argc, char **argv) {

    // initialize the ORB and BOA
    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
    CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

    // Instantiate a math object
    _im_math math_server;

    // Tell the BOA that the object is active and ready
    boa->obj_is_ready(&math_server);

    // Tell the BOA to enter the event-loop
    boa->impl_is_ready();

    return 1;
}
```

Figure 2.5: Server for math object

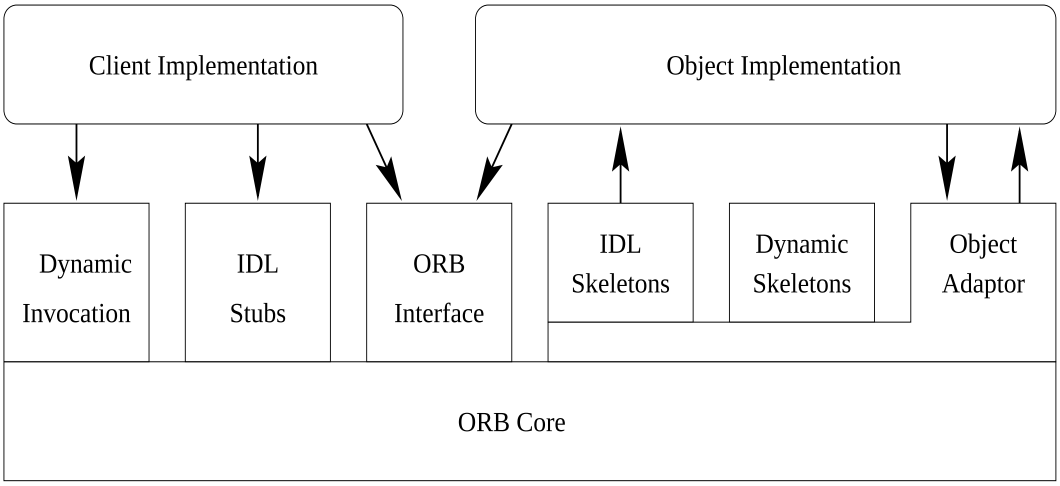

Figure 2.6: The structure of ORB interfaces in CORBA

2.2.1 Support for Object Migration in CORBA

In addition to standardizing the ORB, OMG is also in the process of setting standards for Object Services and Common Facilities. One of the Object Services that has been finalized is the *LifeCycle Service*. The LifeCycle service defines a generic `cosLifeCycle` object that supports operations to move, copy, and destroy objects. Objects that need to provide these lifecycle services should inherit from `cosLifeCycle` interface. The LifeCycle service only defines a client's view of the lifecycle. A client that has an object reference to an object that supports the `cosLifeCycle` interface can invoke operations such as `move()`, `copy()`, and `destroy()` on these objects. The LifeCycle service does not provide any operation to replicate objects. In this thesis, we implement the specifications of the LifeCycle service, which provides for the basic layer to move or copy independent objects. LifeCycle services and their implementations are described in detail in Chapter 4.

Some of the common problems associated with migrating objects are solved in CORBA, because of the underlying transparency provided by the ORB. For instance, once an object is migrated and is registered at the new location, the clients can continue to invoke methods on it. They need not be informed about the new location of the object. Also, as we will see later, CORBA offers convenient mechanisms for keeping track of object locations (using *Naming Service*), and for externalizing the state of the object on the network (using *Externalization Service*).

The basic drawback with CORBA is its inability to support fine-grain objects. Objects in CORBA are large-grain active objects that are bound to a server process. The Interface Definition Language does not have any constructs for defining fine-grain objects which can be exchanged between clients. Further, all the mechanisms needed to exert more control on migration, such as deciding which objects should be migrated, have to be built on the top of the basic functionality provided by the CORBA Object Services. CORBA also does not offer any solutions to the problems of concurrency control, since it does not deal with replication or fine-grain sharing of objects. In this thesis, we build layers on the top of the CORBA Object Services, that give an application developer more control on the objects being migrated, and address the related problems of granularity and concurrency control.

2.2.2 CORBA Compliant ORBs

In addition to the components mentioned above, CORBA also consists of a set of language mappings. These language mappings provide source code compatibility between applications written for different CORBA implementations. Language mappings also define the standards for implementing the basic types of CORBA, and prototypes for the functions that are not part of any CORBA object interfaces. Currently, language mappings exist for C, C++, and Smalltalk. An ORB is *CORBA-compliant* if it follows the CORBA standard for the ORB, and the standards for one language mapping. There are other levels of compatibility defined in CORBA. *Interoperability compliance* requires an ORB to interact with other ORBs using a set of standards (protocols and standard format for object references). It should be noted that Object Services and Common Facilities are not a part of any CORBA compliance definition.

There are many ORBs (both commercial and research oriented) that are CORBA compliant. Examples include Orbeline, HP ORB Plus, and Electra. These ORBs differ in some respects. They use different mechanisms and different function calls for obtaining initial object references. When a client application comes up, it needs to bind to a remote object to obtain its services. Binding involves obtaining the initial object reference. HP ORB Plus uses the `resolve_initial_references()` function and the naming service to obtain the initial references, while Orbeline uses a flat naming scheme, and uses a nonstandard `_bind()` call to obtain the initial reference. Orbeline checks if the objects are located locally on the same node as the client, and bypasses the normal RPC calling style if that is true. Orbeline also provides for some basic fault tolerance features in the form of ability to run multiple instances of `osagent`, which is their standard object location agent, on different nodes. These osagents exchange objects between each other in case of failures.

Electra, on the other hand is an ORB developed primarily to explore the implications of adding group communication, replication, and fault tolerance to the standard CORBA definition. Electra is the ORB that comes closest to our thesis, for having implemented replication. None of these ORBs consider mechanisms for migrating objects of all granularities, grouping objects within a graph, and migrating compound objects. Electra achieves concurrency control by multicasting the messages to all the replicated objects in a group. We propose techniques for replicating parts of the graph, for exerting control on which parts are replicated or migrated, and for fine-grain sharing of the objects within a graph. Further, we support different policies in migration, such as copying, replicating, and completely moving objects and object graphs. Our efforts in this thesis are motivated by a wider set of considerations (which arise from graphics applications) than just fault-tolerance.

2.3 Related Work

There were many attempts to provide transparent operating system support for object migration. Emerald, Guide, and Shadows are some of the examples of these systems.

Emerald [5] supports migration of fine-grain objects. Each object in Emerald has a unique network-wide name, which is used in identifying objects when they are moved between nodes. Objects in Emerald can also have references to other objects. Programming constructs, such as `attached`, are provided to specify which related objects should be moved when one of them moves. It also provides two parameter passing techniques called *call-by-move* and *call-by-visit* which specify whether the parameter object stays at the destination, or is migrated back to the source after the invocation is complete. In addition to the motion of fine-grain objects, Emerald also supports the migration of active objects, along with the associated processes. Before moving the objects, all the processes executing in the object are suspended, a template of the object is made, and the object's state information and template are sent to the new host. The operating system at the new workstation rebuilds the object by allocating space for the object, and copying the state of the object into that space. The template is used in locating the pointers in the state information, which are then replaced with the new addresses. Finally, the processes are resumed on the destination. Emerald uses a *forwarding address* [6] concept to locate objects transparently. The main drawback with Emerald is that it is designed for homogeneous environments and local area networks, and is not scalable to large distributed systems. Only immutable objects can be replicated in Emerald, and the system does not support sharing of stateful objects.

The Guide system [7] allows objects to be migrated at *administration time*, i.e., when there are no methods that are active on the objects. It does not support migration of nonquiescent and active objects. Objects in the Guide system are named by global identifiers which makes them known and sharable in the entire system. An object identifier encapsulates the location of the node on which it was created, which makes an object's location efficient if it is not migrated. Once an object is migrated it leaves a *forwarder* at the source which keeps track of the current location of the object. Shadows [8] is another system that allows mobility of only quiescent objects. Objects move between *object managers* which manage a collection of objects. Location transparency is achieved by a similar forwarding mechanism.

None of the above migration mechanisms consider defining graphs of objects and the issues involved in migrating these graphs. Sharing objects and object graphs is one of the primary requirements of a graphics application, and there is no work which supports this sharing and addresses the related concurrency control issues from the standpoint of a graphics application.

2.4 Object Model

Objects in a graphics application can be large-grain objects (e.g., renderer and model-viewer) or fine-grain objects (e.g., points and lines). The large-grain objects are bound to a server process. In the rest of this thesis, when we talk about large-grain objects, we mean that they are large objects which are bound to a server process (and hence run in a separate address space). A graphics application, typically consists of a few (< 10) large-grain objects and a large ($> 10,000$) number of fine-grain objects.

Two standard *object models* are widely used in the existing systems [9] - the active object model and the passive object model.

In the *active object model* (Figure 2.7), several server processes are created for and assigned to each object to handle its invocation requests. These processes are terminated when the object dies. When a client makes an operation invocation, a process in the corresponding server object performs the operation on the client's behalf. The server object may invoke a method on another object, in which case, a different process is started on the latter object. The process issuing the request waits for the result from the latter process. The number of server processes that are assigned to each object may be either fixed (*static variant of active object model*), or may be dynamically changed when requests arrive at the object (*dynamic variant of the active object model*). Systems such as Amoeba [10] and CHORUS [11] support the active object model.

In the *passive object model* (Figure. 2.8), the processes and objects are separate entities. A process can execute in several objects during its lifetime. When a process makes an invocation on another object, its execution in the object in which it is currently executing is temporarily suspended. It continues its execution in the new object. Emerald [5] and Clouds [12] use the passive object model.

The active object model is suited when the system supports large-grain objects, whereas the passive object model is appropriate when the system supports fine-grain objects. In a system that should support objects of all granularities, we need a *hybrid object model* (Figure 2.9). In such a model, fine-grain objects can reside in the address space of other large-grain objects. The interaction between two objects will depend on the nature of the two objects. If an object A needs to invoke a method on object B, the invocation will follow the passive object model, if B is a fine-grain object, and active object model if B is a large-grain object. This object model encompasses all kinds of objects and is the most suitable one for graphics applications. The advantage of the passive object model is that there is no restriction on the number of processes that can be bound to an object, and it provides a convenient scheme for many objects to coexist and interact with each other. Since most of the objects in a graphics application are fine-grain objects, under our hybrid object model, they interact using the passive object model, thus inheriting all its advantages.

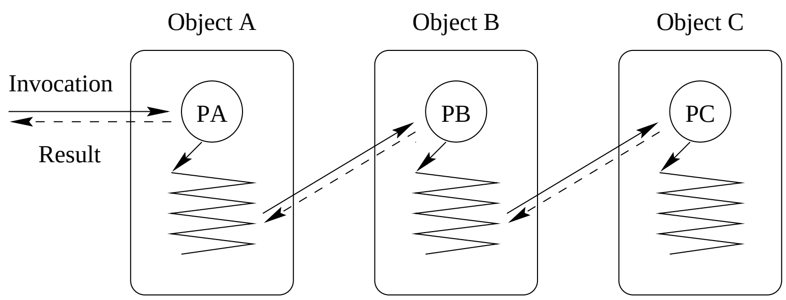

Figure 2.7: Active object model

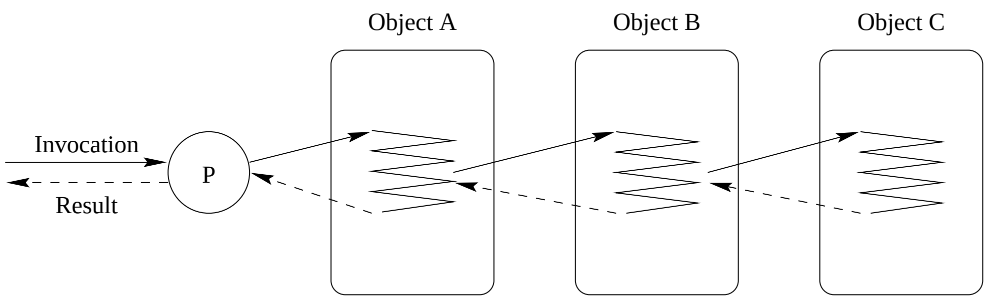

Figure 2.8: Passive object model

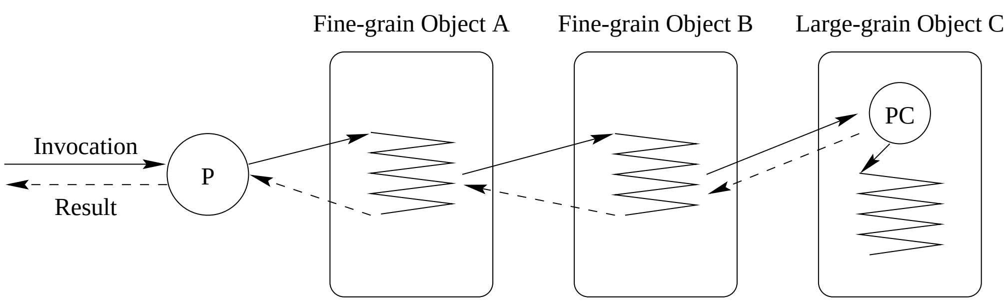

Figure 2.9: Hybrid object model

CHAPTER 3

OBJECT SERVICES

Object Services (Figure 2.2) are a collection of services (interfaces and objects) that provide basic functionality for using and implementing objects. Operations provided by Object Services will serve as building blocks for Application Objects and Common Facilities. Standards have been (and are being) proposed for many Object Services. Two volumes of Common Object Services have been adopted by OMG. The first volume (COSS-1) contains standards for Naming, Event Notification, LifeCycle, and Persistent Object Services. The second volume (COSS-2) describes the standards for Concurrency Control, Externalization, Relationships, and Transactions Services. Standardization work is in progress for Security, Time, Licensing, Properties, Query, Change Management, Collections, and Trader Services.

In this chapter, we discuss the Naming, LifeCycle, and Externalization Services and their implementations. These services form the necessary foundation for implementing object migration. The next chapter shows how these three services can be used in implementing object migration for CORBA objects.

3.1 Naming Service

Naming Service [13] helps in associating objects with names. Objects can be *published* by making their names available through the Naming Service. By *subscribing* to these objects, remote clients can obtain the references to the required objects. Thus, Naming Service provides a convenient mechanism for clients and implementations to exchange object references.

By standardizing the interfaces to the Naming Service, OMG has made it possible for all the clients to use the same methods for obtaining object references. The

IDL declarations for the interfaces to Naming Service are given in the Appendix. Methods are provided for binding, resolving, and unbinding names. All the names are bound with respect to a *context*. A context defines the scope for a binding. No two bindings can have the same name within a context. Contexts are objects themselves, and so they can be bound to names using the Naming Service. This results in a *naming graph* as shown in Figure 3.1. The naming graph is rooted at a context known as the *root context*. To use the Naming Service, a server or a client has to obtain the reference to this root context and invoke a Naming Service operation on it. New contexts can be created by invoking the `new_context()` or `bind_new_context()` methods on a given context. Typically, servers publish their objects in the Naming Service using the `bind()` operation, and clients use the `resolve()` operation to subscribe to these objects before using them. Figures 3.2 and 3.3 demonstrate the use of Naming Service by a server and a client.

3.2 LifeCycle Service

LifeCycle Service [13] defines the interfaces that can be used by clients to invoke lifecycle operations on the CORBA objects. Methods are defined to create, copy, move, and remove CORBA objects. We have extended the LifeCycle Service to include an operation to replicate objects. The modified interfaces are given in the Appendix.

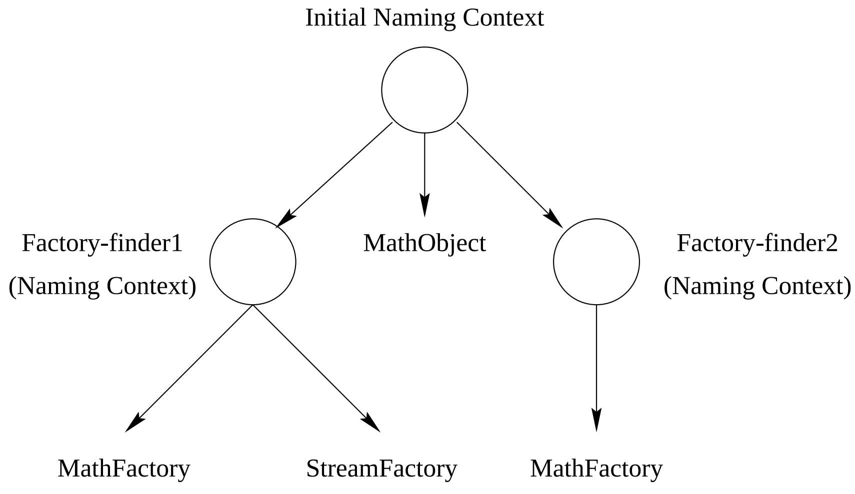

Figure 3.1: Naming graph

```
// server's main
main(int argc, char **argv) {

    // initialize the ORB and BOA
    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
    CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

    // Instantiate a math object
    _im_math math_server;

    // Obtain the reference to the root context in Naming Service
    CosNaming::NamingContext_ptr root =
       CosNaming::NamingContext::_narrow(
          resolve_initial_references( "NameService" ) );

    // publish the object under the name "MathObject"
    CosNaming::Name n;
    n.length(1);
    n[0].id = (const char *) "MathObject";
    n[0].kind = (const char *) "";
    root->bind( n, &math_server );

    // Tell the BOA that the object is active and ready
    boa->obj_is_ready(&math_server);

    // Tell the BOA to enter the event-loop
    boa->impl_is_ready();

    return 1;
}
```

Figure 3.2: Publishing the object in the naming service

```
// Client's main
main(int argc, char **argv) {

    // initialize the ORB
    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

    // Get the reference to the root context in Naming Service
    CosNaming::NamingContext_ptr root =
       CosNaming::NamingContext::_narrow(
          resolve_initial_references( "NameService" ) );

    // Use the naming service and subscribe to the "MathObject"
    CosNaming::Name n;
    n.length(1);
    n[0].id = (const char *) "MathObject";
    n[0].kind = (const char *) "";
    math_ptr m = math::_narrow( root->resolve( n ) );

    // use the math object to invoke methods on it
    m->seta(3.1);
    m->setb(4.3);
    cout << m->add();

    // release the references
    CORBA::release(root);
    CORBA::release(m);

    return 1;
}
```

Figure 3.3: Subscribing to an object using naming service

A client's model of object creation is defined using the notion of *factories*. Factories are just like any other CORBA objects, and their interfaces are defined in IDL. They are used to create other objects. Figure 3.4 shows an example of a math factory. Factories can be located using *factory-finder* objects. A factory-finder is a CORBA object that defines a scope of resource allocation. For instance, a factory-finder can represent a host, a group of machines, or any abstract notion of location. Here, we use factory-finders as analogous to hosts, i.e., each host has one and only one factory-finder associated with it. The `factoryFinder` interface defined in the LifeCycle Service (Appendix) defines only the client's view of a factory-finder. We have implemented a *Naming Context based factory-finder* which inherits from the `NamingContext` interface of the Naming Service and the `factoryFinder` interface of the LifeCycle Service (Figure 3.5).

A factory registers (publishes) itself at the factory-finder running on the same host. Subsequently, clients can contact the factory-finder on the required host to obtain the appropriate factory, and issue a creation request on that factory. Figure 3.6 shows this interaction.

The other lifecycle operations (move, copy, replicate, and remove) have to be provided by the object implementation. Any object that wishes to support these operations can inherit from the CosLifeCycle::LifeCycleObject interface as shown in Figure 3.7, and provide implementations for these operations. To understand the lifecycle operations, it is important to know how clients can invoke these operations on the objects. Figure 3.8 shows a client program that invokes lifecycle operations on the math object declared in Figure 3.7.

The implementation of an object should ensure that all the clients which have a reference to the object should be able to invoke operations on the object, irrespective

```
interface mathFactory {
   math create_object();
}
```

Figure 3.4: Math factory

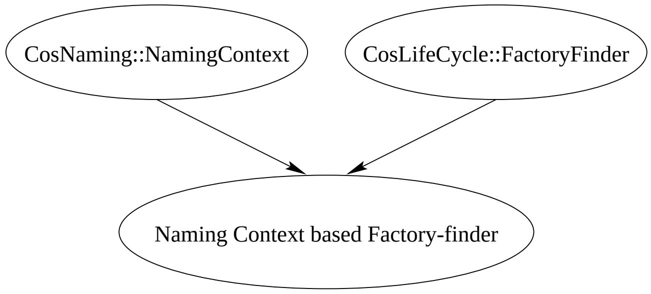

Figure 3.5: Multiple interface inheritance for implementing factory-finders

of the location of the object, and without any knowledge of migration. Notice that the client in Figure 3.8 was able to invoke operations on the math object even after it has been moved, without any other updates. The next chapter explains how objects can support this behavior.

3.3 Externalization Service

In order to provide implementations for the lifecycle operations, objects need to *externalize* their state. Externalization Service [14] defines the standard interfaces for externalizing and internalizing an object's state. The Appendix describes a modified version of the standard IDL specifications for Externalization Service. Any object that wishes to support externalization and internalization operations should inherit its interface from `CosStream::Streamable`, and provide implementations for the two operations `externalize_to_stream()` and `internalize_from_stream()`. *Streams* contain an object's externalized state, and are created as required using a *Stream factory*. The interfaces for streams and stream factories are also described in the Appendix. Figure 3.9 shows how the math object defined in Figure 2.5 can implement externalization operations using stream objects.

The LifeCycle Service and the Externalization Service help an object in supporting migration, while the Naming Service helps the clients in locating the factory-finders which are passed as arguments to the lifecycle operations. These mechanisms are explained in detail in the next chapter.

```
// Client's main
main(int argc, char **argv) {

    // initialize the ORB
    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

    // Get the reference to the root context in Naming Service
    CosNaming::NamingContext_ptr root =
       CosNaming::NamingContext::_narrow(
          resolve_initial_references( "NameService" ) );

    // Use the naming service and subscribe to the FactoryFinder on
    // centauri.cs.utah.edu
    CosNaming::Name n;
    n.length(1);
    n[0].id = (const char *) "centauri.cs.utah.edu";
    n[0].kind = (const char *) "";
    NC_FactoryFinder_ptr nc_ff =
       NC_FactoryFinder::_narrow( root->resolve( n ) );

    // Get the math factory on centauri.cs.utah.edu
    mathFactory_ptr mf =
       mathFactory::_narrow( nc_ff->resolve("MathFactory") );

    // Use the math factory to create a new object on
    // centauri.cs.utah.edu
    math_ptr m = mf->create_object();

    // Use the math object
    m->seta(3.1);

    // release the object references
    CORBA::release(root);
    CORBA::release(mf);
    CORBA::release(m);

    return 1;

}
```

Figure 3.6: Using factory-finders and factories to create objects

```
interface math : CosLifeCycle::LifeCycleObject {
   void seta(in double a);
   void setb(in double b);
   double add();
}
```

Figure 3.7: Math object with lifecycle operations

```
// Client's main
main(int argc, char **argv) {

    // initialize
    // ...

    // create a math object, m, on centauri.cs.utah.edu
    // ...

    // get the factory finder on gemini.cs.utah.edu
    NC_FactoryFinder_ptr gemini_ff = ...

    // Now, move the math object to gemini.cs.utah.edu
    m->move( gemini_ff );

    // Create a replica of m on velo.cs.utah.edu
    NC_FactoryFinder_ptr velo_ff = ...
    math_ptr m_replica = m->replica( velo_ff );

    // use the math objects as usual
    cout << m_replica->add();

    // release the object references
    // ...

    return 1;

}
```

Figure 3.8: Client invoking lifecycle operations on math object

```
void _im_math::internalize_from_stream(
   CosStream::Stream_ptr sourceStreamIO,
   CosLifeCycle::FactoryFinder_ptr there)
   throw (
      CosLifeCycle::NoFactory,
      CosStream::ObjectCreationError,
      CosStream::StreamDataFormatError)
{
   x = sourceStreamIO->read_double();
   y = sourceStreamIO->read_double();
}

void _im_math::externalize_to_stream(
   CosStream::Stream_ptr targetStreamIO)
{
   targetStreamIO->write_double(x);
   targetStreamIO->write_double(y);
}
```

Figure 3.9: Externalization operations implemented by the object

CHAPTER 4

IMPLEMENTING OBJECT MIGRATION

In this chapter, we discuss how objects can be made to support migration when they receive the appropriate requests from the clients. This discussion is mainly classified into two sections. The first one explains how large-grain objects can implement object migration. The second one talks about the fine-grain objects. Large-grain objects are declared in IDL, and are implemented as CORBA servers, whereas fine-grain objects are declared and defined in an object-oriented language (C++, in our case), and remain in the address space of the client. Since these objects are not declared in IDL, they cannot be directly recognized by the CORBA ORB. The section on fine-grain objects describes how one can solve this problem.

4.1 Large-grain Objects

The object services described in the previous chapter can be used to implement object migration for large-grain objects. Any object that wishes to support lifecycle operations should inherit from `CosLifeCycle::LifeCycleObject` and `CosStream::Streamable`, and provide implementations for the methods in these interfaces. The complete IDL specification for a math object that supports migration is shown in Figure 4.1.

Figure 4.2 shows how this object can be made to support the `copy()` operation. As parameters to the copy operation, the math object receives the factory-finder that corresponds to the destination. The math object uses the factory-finder to locate a math factory at the destination. It then issues a `create_object()` request on the math factory and obtains a reference to the newly created math object. It externalizes its own state into a stream object and allows the new math object to internalize this state from the stream object.

```
interface math : CosLifeCycle::LifeCycleObject,
                 CosStream::Streamable
{
   void seta(in double a);
   void setb(in double b);
   double add();
}
```

Figure 4.1: Math object supporting migration

```
CosLifeCycle::LifeCycleObject_ptr _im_math::copy(
   CosLifeCycle::FactoryFinder_ptr there,
   const CosLifeCycle::Criteria& the_criteria)
{
   // get a stream object
   CosStream::Stream_ptr s = GetStream();

   // externalize the state into the stream object
   externalize_to_stream(s);

   // use the factoryfinder to create a new object
   mathFactory_ptr f = mathFactory::_narrow(
      GetFactory("mathFactory",
                 NC_FactoryFinder::_narrow(there)));
   math_ptr new_o = f->create_object();

   // ask the new object to internalize the state
   new_o->internalize_from_stream(s, NULL);
   s->remove();
   return new_o;
}
```

Figure 4.2: Implementation of the copy operation

The implementation for `replicate()` is similar to that of `copy()` except that the object needs to store certain information about its replicas. This information can be used to bring the objects into synchronization when required. It should also be noted that all the earlier replicas of the math object need to be informed about the newly created replica. This can be done by the `add_replica()` operation provided in the LifeCycle Service (Appendix).

Moving an object involves copying the object to the destination, and deleting the current object. All the clients having a reference to the target object should be able to use the object even after migration. This can be done in ORBeline, using the *rebind-enable* feature. With this feature, when a client detects that the target object no longer exists, it takes the help of ORBeline to bind to a new object having the same *ORBeline name*. ORBeline names are names that can be assigned to objects at the time of creation. We exploit this feature of ORBeline and implement the move operation by creating a new object at the destination with the same ORBeline name. Clients will be unaware of the rebind mechanism, and all the calls will be directed to the new target (after rebind) without any *forwarding* or *residual dependency*. Once the state has been transferred from the source object to the target object, the source can terminate itself. The ORB will automatically route the pending requests at the source object to the new location.

Objects also need to provide an implementation for `externalize_to_stream()` and `internalize_from_stream()` operations, as described in the last chapter, for the above operations to work successfully. For the case of single objects which do not depend on any other objects (which do not have pointers to other objects), providing implementations for the externalization operations is straight forward. The case when objects can contain pointers to other objects (resulting in a graph of objects) is discussed in the next chapter.

All the services described in the previous chapter are implemented as distinct CORBA servers. These servers need to be running on all the machines which participate as the source or destination of any migration operation.

4.2 Fine-grain Objects

Fine-grain objects pose special problems as they are not supported by CORBA. CORBA can only recognize objects whose interfaces are declared in IDL. It would not be feasible to declare all the fine-grain object interfaces in IDL due to several reasons. Firstly, all the existing C++ applications have to be restructured by changing their C++ class declarations into IDL. It would be very difficult to convert a C++ class containing pointers to objects of other classes into corresponding IDL. Further, there are no parallels in IDL for many features of C++ such as implementation inheritance, pure virtual functions, and virtual inheritance. The second and the more important reason is that by declaring an object's interface in IDL, the object needs to be implemented as a CORBA server. This defeats the primary purpose of having fine-grain objects which are required to share the address space of the application using them.

So, we need a mechanism in which fine-grain objects can be created in the application, but are still visible to the remote clients. If a remote client wants to invoke any operations on these objects, it can first migrate them into its own address space before using them. Such a model has the benefits of fine-grain interaction within a client while still allowing the objects to be distributed. We first present an application's view of creation and use of these fine-grain objects. Then, we describe how fine-grain objects can be made to support such behavior.

Figure 4.3 shows an application creating a fine-grain object of type `pt`. The C++ declaration for the `class pt` is also shown in the figure. Notice that this class inherits from `fgObject<pt>`. This is elaborated later. A `pt` object is constructed using the normal C++ constructor mechanisms, which makes the created object part of the address space of the application.

Remote applications need to be able to migrate (copy, replicate, or move) this object into their address space. Since, the reference to this object is not visible to remote applications, we need to have a *representative object* which is a CORBA object, and which acts as the representative for the `pt` object. Remote applications can now contact the representative to get the desired services. The fact that their

```
// class pt is a normal C++ class
class pt : public fgObject<pt> {
public:
   pt() {};
   ~pt() {};
   double x;
   double y;
};

main()
{
   // Initialize the ORB
   ...

   // Instantiate a pt object
   pt *pt_obj = new pt;
   pt_obj->x = 1.0;
   pt_obj->y = 2.0;

   // Publish the pt object under the name PTObject
   pt_obj->publish(``PTObject'');

   // do the rest of the stuff
   ...
}
```

Figure 4.3: Creation of fine-grain objects

services are being serviced by the representative is completely transparent to the remote applications.

Every fine-grain object has its own representative. The representative objects come into picture only when an object needs to be migrated from one application to another. Otherwise, the objects within an application interact in a fine-grain manner. An application that wishes to allow its objects to be migrated to a different application *publishes* its objects using the naming service. Publishing a fine-grain object involves publishing its representative in the Naming Service. Figure 4.4 shows a remote application subscribing to a fine-grain object and migrating it into its address space.

```
class pt : public fgObject<pt> {
public:
   pt() {};
   ~pt() {};
   double x;
   double y;
};

main()
{
   // Initialize the ORB
   ...

   // subscribe to the PTObject
   representative_ptr r = pt::subscribe(``PTObject'');

   // move the remote object to this address space
   pt *pt_obj = pt::_narrow(r->move());
   cout << pt_obj->x << ', ' << pt_obj->y << endl;

   // do the rest of the stuff
   ...
}
```

Figure 4.4: Moving a fine-grain object

4.2.1 Representative Objects

As explained earlier, representative objects help the remote applications access fine-grain objects from another application. An application requests the representative object to `copy()`, `move()`, or `replicate()` the associated fine-grain object. From then onwards, the application and the representative object cooperate to create a new fine-grain object in the requesting application, and to transfer the state of the source fine-grain object to the destination fine-grain object. This interaction is explained below:

1. An application subscribes (using `subscribe()`) to a representative object and invokes the appropriate migration operation on it (`move()` in Figure 4.4).

2. The representative object sends a message to the source object to externalize its latest state into a stream.

3. The application that initiated the migration, creates a new object in its address space, and internalizes the above state into the new object. This is accomplished by the `_narrow()` operation in Figure 4.4.

In step-2 above, the representative object needs to contact the associated fine-grain object to invoke an externalization operation. This is accomplished by *callback objects*. Each application is associated with *one* callback object. This callback object can accept requests from the representative objects on behalf of the fine-grain objects in the application. All the fine-grain objects in the application register at the callback object at creation time. These steps are summarized below:

1. An application initializes a callback object that can accept requests on behalf of all its fine-grain objects. The callback object should be waiting for requests from representatives, and hence it is started in a separate thread of execution within the application as a normal CORBA server.

2. Any object that is created by the application should register itself with the callback object. This involves assigning an unique name (unique within the

application) to the object. Registration constitutes a promise by the callback object that it will forward all the requests received from representative objects to the appropriate fine-grain objects.

3. For every object that is created by the application, a representative object is also created (as discussed earlier). This can be done using a *representative object factory*. The representative object should have sufficient information (a reference to the callback object at which the associated fine-grain object is registered, and the unique name of the associated fine-grain object) to be able to send a message to the fine-grain object when required. Sample interfaces for a representative object and for a factory that creates these objects are shown in Figure 4.5.

All these interactions are shown in Figure 4.6. When the object is deleted in an application, it should be unregistered from the application's callback object, and its representative object should be deleted. Figure 4.7 shows the IDL for a callback object which is implemented as a CORBA server in each application. In addition to the methods shown in Figure 4.7, a callback object should support methods to register and unregister objects. Note that these methods need to be available only within the application in which the callback object was created. Hence, they do not appear in the IDL of the callback object. A skeleton implementation for the callback object is shown in Figure 4.8. The initialization that every application should go through, in terms of starting a callback object in a separate thread, is shown in Figure 4.9. Since multiple threads can be accessing the fine-grain objects simultaneously, care should be taken to ensure proper coordination between threads. This could be done using *mutual exclusion* variables.

In addition to the normal operations that should be supported, every object should support the following operations:

- `publish()` - publish an object's representative in the Naming Service.

```
interface representative {
   // this is the reference to the callback event handler
   attribute callback event_handler;

   representative copy();
   void move();
   representative replicate()
   void remove();
};

interface representativeFactory {
   representative create_object();
};
```

Figure 4.5: IDL for representative objects and their factories

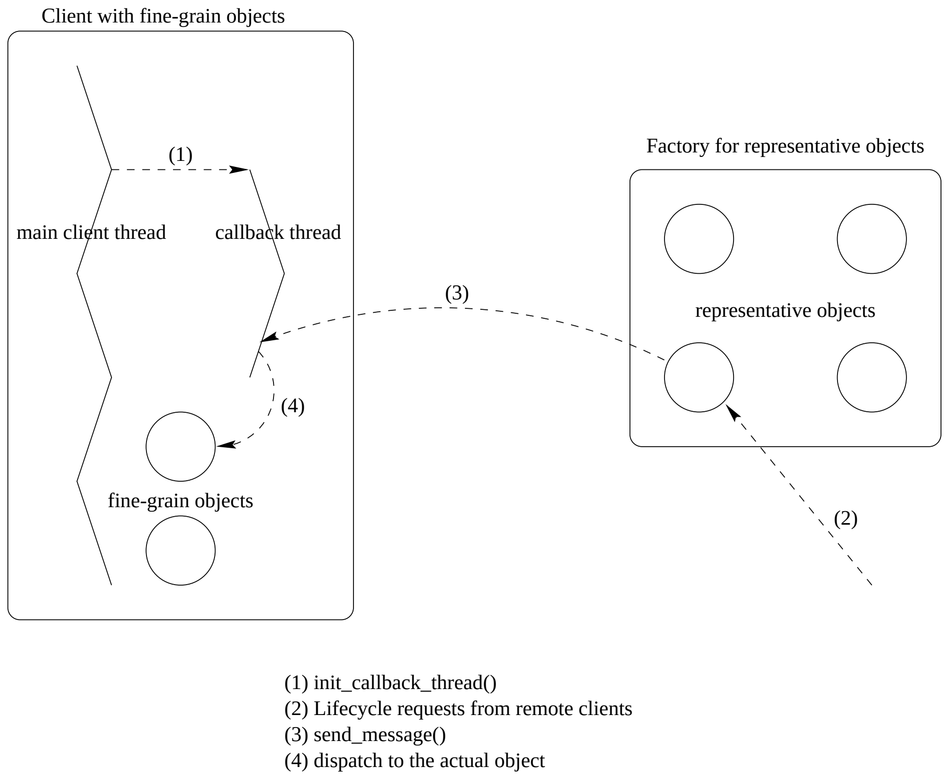

Figure 4.6: Representative and callback objects

```
interface callback {
   void send_message(in string obj_name, in string op_name,
                     inout CosStream::Stream s, inout Traversal t);
};
```

Figure 4.7: IDL for a callback object

```
class _im_callback : public _sk_callback
{
   // private data members ...
public:
   _im_callback();
   ~_im_callback();

   // this method is exported to representative objects
   void send_message(const char * obj_name,
                     const char * op_name,
                     CosStream::Stream_ptr& s,
                     Traversal_ptr& t);

   // the following methods are used by the objects within the
   // application to register and unregister at the callback object.
   void register_object(void *obj_ptr, const char *obj_name);
   void unregister_object(const char *obj_name);
};
```

Figure 4.8: Implementation of a callback object

```
void *callback_thread(void *arg)
{
   _im_callback fg_handler;

   // Tell the BOA that the object is ready
   boa->obj_is_ready(&fg_handler);

   // Event loop
   boa->impl_is_ready();

   return NULL;
}

main(int argc, char **argv)
{
   // Initialize the ORB
   CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
   // Initialize the BOA
   CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

   // start of a new thread for callback handler
   thread_t tid;
   if (thr_create(NULL, NULL, callback_thread, NULL, NULL, tid)) {
      cerr << "Could not start a callback thread" << endl;
      exit 1;
   }

   // do the rest of the stuff
   ...
};
```

Figure 4.9: Initialization of the callback object in an application.

- `subscribe()` - subscribe to an object that was published earlier in the Naming Service.
- `unpublish()` - remove the object from the Naming Service before deleting the object.
- `_narrow()` - create a new object and internalize the state into it from a stream object.

These operations are common to every fine-grain object that can be created by the application. Hence, these operations are provided by a template class `fgObject` which constitutes the base class of all other classes declared in the application. A simplified interface for `class fgObject` is shown in Figure 4.10.

All the objects discussed in this chapter were single objects without any pointers to other objects. Objects in a graphics system do not exist in isolation. We waive the restriction of isolated objects and see how graphs of objects can be made to support migration, in the next chapter.

```
template <class T>
class fgObject
{
   // private data members
public:
   fgObject();
   ~fgObject();

   void publish(const char *n);
   void unpublish();
   static representative_ptr subscribe(const char *name);

   static T *_narrow(CORBA::Object_ptr obj);
   static T *_create(int obj_count = 1, T *t_obj = NULL);
};
```

Figure 4.10: IDL for a callback object

CHAPTER 5

OBJECT GRAPHS AND GRANULARITY CONTROL

Objects in a distributed system do not exist in isolation. In general, they will be related to other objects, resulting in a graph of objects. These graphs introduce special problems for migration. We consider the graph in Figure 5.1 as an example for the following discussion. This graph is composed of objects of two types - `A` and `B`. A1, A2, A3, and A4 are objects of type `A`, while B1 and B2 are objects of type `B`. Every `A` object points to two other objects - one `A` object and one `B` object. Every `B` object points to one `A` object. The *leaves* of the graph are an exception to this rule, and they may contain NULL pointers.

- When an object in a graph is migrated, the objects that it points to should also be migrated. In Figure 5.1, when object A1 is migrated, objects A2 and B1 should also be migrated so that the two pointers of A1 point to valid objects when they are dereferenced. This means that the entire graph has to be reconstructed at the destination.

- Graphs in a graphics system are often composed of large number of objects. It would not be feasible to migrate the entire graph in one step, as mentioned above. In certain cases, it would be better to migrate only a portion of the graph, leaving the rest of the graph at the source. The remaining portion could be migrated as and when it is required at the destination. This introduces the notion of *migration policies* which address the issues of granularity control over migration of graphs.

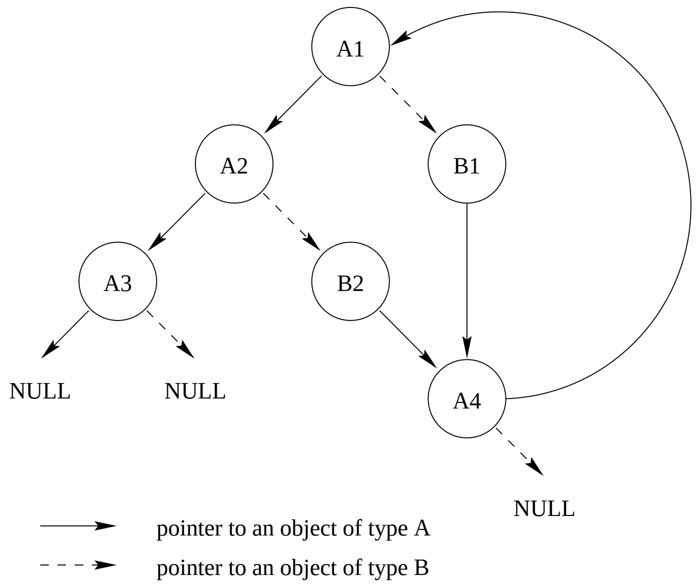

Figure 5.1: A graph of objects

- Objects can be *shared* in an object graph. For example, object A4 is shared by objects B1 and B2. Special care should be taken to preserve the same sharing semantics at the destination.

- Classes in C++ have many features such as simple or multiple inheritance, and virtual inheritance. Further, memory for objects can be dynamically allocated. These language-specific features pose special problems for implementing migration of graphs of objects.

This chapter discusses some of the solutions to these problems, and explains how we have implemented migration of object graphs. This chapter also introduces the notion of *smart pointers* which would help in an efficient implementation of all the problems discussed above. We first start with a very simple model for implementing migration for graphs of objects. In addition to providing more insight into the issues involved, this model provides a basis for extending the model, in subsequent sections.

5.1 Simple Model for Migration of Graphs

Figure 5.2 shows the declarations for the classes `A` and `B` mentioned in Figure 5.1. We assume that each pointer in these classes is either a NULL pointer or points to a *single object*, that may be dynamically allocated. In particular, these pointers to do not point to an array of objects. We also assume that objects are not shared within a graph. The code in Figure 5.3 shows an implementation for the externalization and internalization operations on the objects of these classes. Having defined the externalization and internalization operations in this fashion, the way in which clients migrate the object graph is exactly similar to the model discussed in Figures 4.3 and 4.4 of the last chapter. Applications subscribe to an object (within the graph) that was published earlier by a different application in the Naming Service, and initiate a migration operation on it. The implementation of the migration operation initializes the externalization operation on the object (through representative and callback objects). This would result in the entire subgraph rooted at that object to be externalized into the stream. At the destination the object graph is reconstructed by the internalization operation. Thus, we notice that a proper definition of the `externalize_to_stream` and `internalize_from_stream` operations results in a support for simple object graph migration without any further modifications to the rest of the system. In the next section, we take a look at the migration policies that control the granularity of migration.

5.2 Migration Policies

In some cases it would be preferable to migrate the entire graph of objects at one time. For instance, when we are rendering a model graph, it would be advisable to migrate the entire model graph to the rendering application, since all the objects in the graph would be used by the application immediately. However, in some cases, it is not necessary to migrate the entire graph to the destination. Consider an application that requires access to only a subgraph of the entire model graph. Such situations arise very frequently in design environments where users work on parts of the model instead of working on the entire model. In these cases, we just

```
class B;

class A : public fgObject<A>
{
public:
   A *ap;
   B *bp;

   A();
   ~A();

   virtual void externalize_to_stream(CosStream::Stream_ptr s);
   virtual void internalize_from_stream(CosStream::Stream_ptr s);
};

class B : public fgObject<B>
{
public:
   A *ap;

   B();
   ~B();

   virtual void externalize_to_stream(CosStream::Stream_ptr s);
   virtual void internalize_from_stream(CosStream::Stream_ptr s);
};
```

Figure 5.2: Classes `A` and `B`

```
void A::externalize_to_stream(CosStream::Stream_ptr s)
{
   if (ap == NULL)
      s->write_long(0);
   else {
      s->write_long(1);
      ap->externalize_to_stream(s);
   }

   // similarly for bp ...
}


void A::internalize_from_stream(CosStream::Stream_ptr s)
{
   int code = s->read_long();
   if (code == 0)
      ap = NULL;
   else {
      ap = new A;
      ap->internalize_from_stream(s);
   }

   // similarly for bp ...
}

// similarly for class B
// ...
```

Figure 5.3: Simple externalization and internalization operations for graphs

need to migrate the required subgraph. The rest of the graph can be migrated only when required (perhaps to merge the part with the rest of the model). The necessary granularity control that is required in migrating subgraphs within a graph is specified by *migration policies*.

Two simple migration policies are *shallow migration* and *deep migration*. Under the shallow migration policy, only one object is migrated at a time. The other objects that it refers to are migrated only when they are required (when the corresponding pointers are dereferenced). Under the deep migration policy, all the objects that can be reached from the root object (the object on which the migration operation was invoked) are migrated immediately. The code shown in Figure 5.3 implements a deep migration policy. In most of the graphics applications, a need arises for a range of migration policies between these two extremes. The richer the range, the more control we have on the granularity of migration. The rest of this section discusses some of possibilities that we have provided in our implementation to enrich the range.

5.2.1 Depth of Migration

One can migrate all the objects that can be reached from the root within a certain depth. For example, if we invoke a migration operation on A1 in Figure 5.1 with a depth of 2, the objects A1, A2, and B1 will be migrated. When the pointers in these objects are dereferenced at the destination, another set of objects within a depth of 2 from those objects will be migrated. For instance, if the pointer to B2 was dereferenced in object A2, objects B2 and A4 will be migrated into the destination. Figure 5.4 demonstrates this policy.

This migration policy can be used to simulate the effects of *neighborhood of an object* within a graph. A neighborhood of an object comprises of all the objects that are either dependent on the object or are prerequisites for the construction of the object, up to a certain depth. Neighborhood is a powerful granularity control paradigm that is often used in the design of models in Alpha_1.

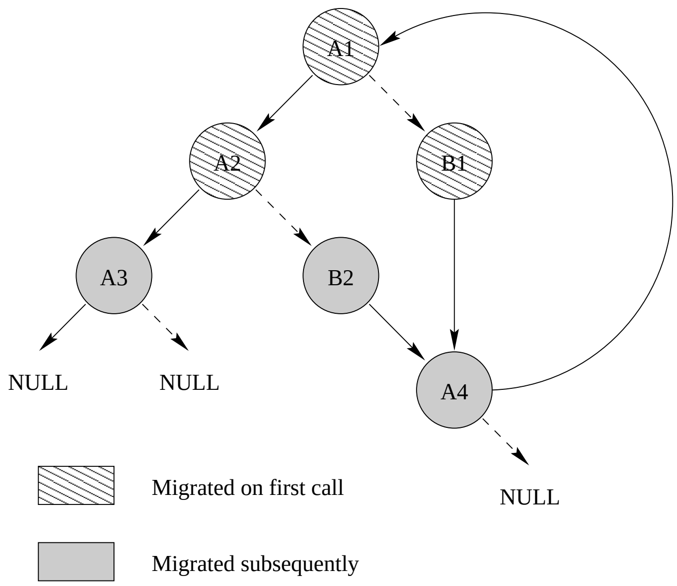

Figure 5.4: Migration policy based on depth

5.2.2 Class Boundaries

Sometimes the application demands that object of particular types be migrated whereas objects of other types be left at the source. For example, a rendering application might render the model only up to the level of detail of polygons. Other objects such as points and lines may not be rendered by the application. In this case, all the objects in the model graph can be migrated until we hit upon a line object or a point object. The subgraph that is rooted at the point and line objects need not be migrated.

Considering our earlier example, if we set a class boundary of migration at objects of class `B`, objects A1, A2, and A3 will be migrated initially. The rest of the graph will be migrated when it is required. Figure 5.5 shows this behavior.

A variant of this policy is the *diffusion model*, in which each class is associated with a *migration power* between 0 and 1. The migration is initiated with an initial *power* of 1. When an object of a class is migrated, the initial power decreases by the power of class. When the initial power decreases to a value of 0, the migrates stops. Figure 5.6 demonstrates this policy with a power of 0.6 associated with objects of class `A` and a power of 0.3 associated with objects of class `B`.

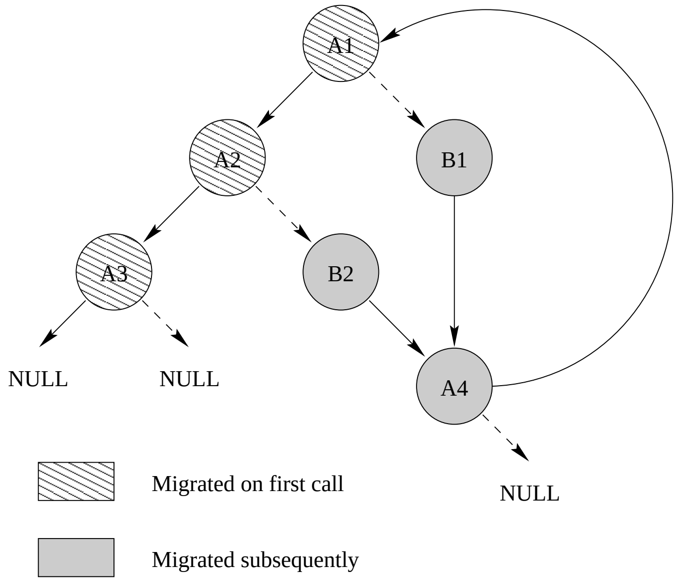

Figure 5.5: Migration policy based on class boundaries

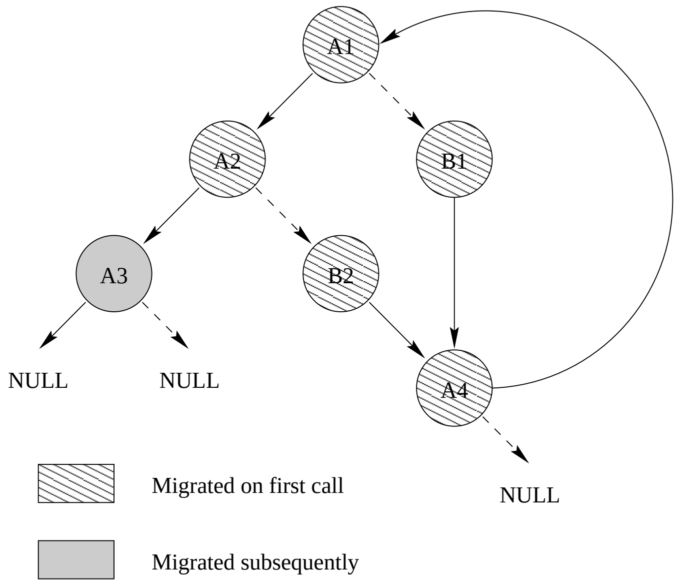

Figure 5.6: Migration policy based on diffusion model

5.3 Smart Pointers

Having considered different migration policies, we now discuss a method called *smart pointers* for implementing these policies. Smart pointers will also be useful in solving some problems associated with language-dependent features, as will be described in the last section of this chapter.

Smart pointers are special objects for which the dereferencing operator (`-> operator`) is overloaded [15]. The overloaded operator performs the extra functionality required to migrate the necessary objects into the address space of the destination, before returning a pointer to the required object. Figure 5.7 shows a simple implementation for the smart pointer.

In addition to the `->` operator, other dereferencing operators such as `*` (star) and `[]` (array indexing) should also be overloaded. To ensure the full functionality of a normal pointer, other operations such as pointer comparison, pointer arithmetic, and certain type cast operators need to be defined for smart pointers.

Each smart pointer keeps track of the actual pointer to the object, if the object

```
template<class T>
class smartPointer
{
   // some private data
public:
   // ...
   T *operator->();
   T& operator*();
   T& operator[](int i);

   // overloaded operations
   smartPointer<T>& operator= (T *t_ptr);
   operator T *();
   int operator== (T *t_ptr);
   int operator!= (T *t_ptr);
   ...
};
```

Figure 5.7: Smart pointers

is within the address space of the application. If it is not in the application's address space, it stores sufficient information to be able to migrate the object from the source, when one of the dereferencing operations is invoked on it. With the introduction of the smart pointers, all the classes have to be redeclared by replacing the normal pointers with smart pointers. A modified version of the classes `A` and `B` is shown in Figure 5.8.

```
class B;

class A : public fgObject<A>
{
public:
   smartPointer<A> ap;
   smartPointer<B> bp;

   A();
   ~A();

   virtual void externalize_to_stream(CosStream::Stream_ptr s,
                                      Traversal_ptr t);
   virtual void internalize_from_stream(CosStream::Stream_ptr s,
                                        Traversal_ptr t);
};

class B : public fgObject<B>
{
public:
   smartPointer<A> ap;

   B();
   ~B();

   virtual void externalize_to_stream(CosStream::Stream_ptr s,
                                      Traversal_ptr t);
   virtual void internalize_from_stream(CosStream::Stream_ptr s,
                                        Traversal_ptr t);
};
```

Figure 5.8: Classes `A` and `B` with smart pointers

Before showing how to implement the externalization operations that incorporate migration policies, we introduce *traversal objects* which are useful in controlling (setting and changing) these migration policies. A traversal object keeps track of the current migration policy and supports methods to change these settings. The IDL for a traversal object is shown in Figure 5.9. The overloaded `migration_type()` methods can be used to query and set the migration policy. The `propagation_power` attribute is used in implementing class boundaries and diffusion model explained in section 5.2. The traversal object is passed as a parameter to all the externalization and internalization requests (see Figure 5.8).

To implement the migration policies, the methods `externalize_to_stream()` and `internalize_from_stream()` have to be changed as shown in Figures 5.10 and 5.11. Before externalizing a smart pointer, a check is first made to see if the pointer is a NULL pointer. If it is not, the object pointed to by the pointer is externalized, if the migration policy permits it. In the case of Figure 5.10, the object pointed to by `ap` is externalized if the migration policy is *deep* or if there is enough power (diffusion power of the class) to migrate the object. If both these conditions fail, the object is not externalized. Instead, the reference to its representative object is written out to the stream. The internalization routine can store this reference (Figure 5.11), and use it to invoke another migration operation when the corresponding smart pointer is dereferenced.

```
interface Traversal {

   enum MigrationMode {move, copy, replicate};
   enum MigrationType {shallow, deep, type_specific};

   attribute MigrationMode migration_mode;
   attribute double propagation_power;
   attribute MigrationType migration_type;
   oneway void remove();
};
```

Figure 5.9: Traversal objects

```
void A::externalize_to_stream(
   CosStream::Stream_ptr s, Traversal_ptr t)
{
   Traversal::MigrationType _migration_type = t->migration_type();
   double _remaining_power = t->propagation_power();

   if (ap == NULL)
      s->write_long(0);
   else if (((_migration_type == Traversal::deep) ||
            ((_migration_type == Traversal::type_specific) &&
             (_remaining_power >= a::_g_power)))) {
      s->write_long(1);
      t->propagation_power(_remaining_power - ap->_g_power);
      ap->externalize_to_stream(s, t);
   }
   else {
      s->write_long(2);
      // write the reference for the representative object
      // This will be stored by the smart pointer at the destination
      // and will be used to migrate the object when required
      s->write_object(ap._fp);
   }

   // similarly for bp ...
}
```

Figure 5.10: Externalization operation with migration policies

```
void A::internalize_from_stream(
   CosStream::Stream_ptr s, Traversal_ptr t)
{
   long _index = s->read_long();
   if (_index == 0)
      ap = NULL;
   else if (_index == 1) {
      if (ap == NULL)
         ap = fgObject<a>::_create();
      ap->internalize_from_stream(s, t);
   }
   else
      // the object should be migrated later
      ap._fp = fg::_narrow(s->read_object());

   // similarly for bp ...
}
```

Figure 5.11: Internalization operation with migration policies

5.4 Shared Objects in a Graph

An object in a graph is said to be shared if there is more than one object that refers to that object, or if it is part of a cycle within the graph. From the example in Figure 5.1, objects A1 and A4 are shared. If we follow the pattern in Figures 5.10 and 5.11 to externalize an object, the shared objects will be externalized multiple times. Special care should be taken if an object was already externalized before writing it to the stream. This could be done once again using the traversal object. Each traversal object keeps track of which objects were externalized into the stream. It maintains a mapping between the object names (internal ORBeline names) and unique index numbers that are assigned to those objects. One can query the traversal object to find out whether an object was already externalized, and if so, get its index number. Instead of writing the object multiple times to the stream, the index number is written. At the destination, the application that initialized the migration request keeps track of a mapping between the index numbers and the pointers to the actual objects that were created within the application. These mappings can be used to reestablish the same sharing semantics that were present at the source.

5.5 Language Dependent Features

Some of the complications in externalizing an object resulting from the features of the C++ language are discussed in this section. In particular, we look at the problems posed by dynamic memory allocation for objects, inheritance, and multiple level of pointers to objects.

In C++, objects can be dynamically allocated using the `new` operator. Further, a pointer to an object can be made to point to an array of objects using the `new []` operator. As a result, the number of objects that a pointer points to cannot be known at compile time. Also, there is no standard way to find the number of objects in the allocated array using a library routine in C++. This complicates the generation of code for the externalization and internalization routines. Once again, the notion of smart pointers helps in solving this problem. Since we have replaced all the pointers in the C++ class declarations with smart pointers, we can make the smart pointers keep track of the number of objects allocated dynamically. This can be used to generate the appropriate code for externalization and internalization routines. The changes required in the declaration of the smart pointers, the creation routines used in dynamically creating arrays of objects (instead of `new []`), and the enhancements to the externalization routines for the examples in this chapter, are shown in Figure 5.12.

Classes in C++ can inherit from other classes. There are several flavors of inheritance in C++ including simple inheritance, multiple inheritance, and virtual inheritance. To externalize an object whose class (say Y) is inherited from another class (say X), we need to externalize the data members declared in both X and Y. This rule applies to all the cases of simple and multiple inheritance, and is demonstrated in Figure 5.13. It does not cause any harm to extend the same strategy to virtual inheritance as long as the code to internalize an object is compatible with the code to externalize it.

```
// changes to smart pointers
template <class T>
class smartPointer {
   int _size;          // size of the array of objects
   T *_ptr;            // pointer to the array
   // rest of the declaration is same as shown earlier.
};

template <class T>
class fgObject {
   // ...
public:
   // ...
   // _create is equivalent to new [] in C++
   smartPointer<T> _create(int num_objects);
};

// enhancements to externalization routine for class A
void A::externalize_to_stream(
   CosStream::Stream_ptr s, Traversal_ptr t)
{
   // ...
   if (ap == NULL)
      s->write_long(0);
   else if (((_migration_type == Traversal::deep) ||
            ((_migration_type == Traversal::type_specific) &&
             (_remaining_power >= a::_g_power)))) {
      s->write_long(1);
      // number of objects in the array
      s->write_long(ap._size);
      t->propagation_power(_remaining_power - ap->_g_power);
      for (int i = 0; i < ap._size; i++)
         ap[i].externalize_to_stream(s, t);
   }
   else {
      // ...
   }

   // similarly for bp ...
}
```

Figure 5.12: Changes to incorporate arrays of objects

```
class X
{
   // data members in X
};

class Y : public X
{
   // data members in Y
};

void X::externalize_to_stream(CosStream::Stream_ptr s,
                              Traversal_ptr t)
{
   // code to externalize data members in X
}

void Y::externalize_to_stream(CosStream::Stream_ptr s,
                              Traversal_ptr t)
{
   // first externalize the data members in X
   X::externalize_to_stream();

   // code to externalize data members in Y
}
```

Figure 5.13: Inheritance and externalization routines

Finally, pointers need not be defined at a single level in C++. Figure 5.14 shows the different ways in which pointers can be defined in a C++ class. Smart pointers, as defined earlier, need no further modifications to incorporate these cases. Figure 5.14 shows how smart pointers can be nested to solve this problem.

```
class test;
class different_pointers
{
   test *p1;
   test **p2;
   test ***p3;
   test *p4[10];
   test **p5[10][20];
};

// the above class is equivalent to the following class
// using smart pointers
class different_pointers
{
   smartPointer< test > p1;
   smartPointer< smartPointer< test >> p2;
   smartPointer< smartPointer< smartPointer < test >>> p3;
   smartPointer< test > p4[10];
   smartPointer< smartPointer< test >> p5[10][20];
};
```

Figure 5.14: Multiple levels of pointers and their smart pointer equivalents

CHAPTER 6

CONCURRENCY CONTROL

One of the ways for migrating an object, as described in Chapter 2, is replication. The replicated object and the target object share the same logical identity. This implies that when one of the replicas is modified, it should be reflected in the other replicas as well. Replication of objects provides a simple and convenient paradigm for the sharing of objects between different applications. However, it also introduces new problems in terms of concurrency control. Simultaneous accesses to the replicas might result in inconsistent modifications on the same object. This chapter addresses the concurrency control issues that arise from replication of objects and the appropriate strategies to solve these problems from the stand point of graphics applications.

6.1 Transaction Models

In order to maintain consistency of data in an application, certain application-specific consistency requirements must be satisfied. For example, in a banking application performing a money transfer, the amount of money debited from an account should be equal to the amount of money credited in another. The application-specific consistency requirement in this case is that the total sum of the money in the associated accounts should be constant. The required consistency requirements are satisfied by grouping the series of instructions performing the money transfer into a sequence called *transaction*. Assuming that the data manipulated by the application resides in a database, users maintain consistency in the data by performing transactions on the database. Transactions play three distinct and important roles in maintaining the consistency of data [16]: (a) they are logical units that group together operations comprising a complete task, (b) they are *atomicity* units whose execution preserves the consistency of the database, and (3) they are recovery units that ensure that either all the steps enclosed within them are executed or none are. Thus, by definition, a transaction takes a database from one consistent state to another.

When multiple users execute transactions simultaneously on the database, we need a concurrency control strategy that maintains the required consistency inspite of concurrent accesses. We can say that the database is in a consistent state if the net effect on the database is identical to the case when each of the transactions executed one after the other in any particular order. This additional requirement in lieu of concurrent transactions is known as *serializability.* Most of the concurrency control strategies revolve around the notion of serializability in maintaining database consistency. For example, they would not allow any transaction that might jeopardize the serializability requirement, or they might *rollback* a transaction that violates the serializability requirement. It should be noted that serializability is not a necessary condition for ensuring consistency of concurrent transactions. A *schedule* of transactions might not be serializable, but still satisfy application-specific consistency requirements.

Some of the standard concurrency control techniques include the locking protocols (two-phase locking being the most popular), timestamp ordering, and optimistic concurrency control schemes [16]. We assume that the reader is familiar with the concepts in some of these concurrency control strategies. To clearly understand the drawbacks of these schemes when applied to graphics applications, and for the rest of the discussion in this chapter, we consider an example. Figure 6.1 shows a simple model graph showing the design of a car. The components in the design are shown as nodes in the graph. An arc from one node to another indicates a dependency between the associated components. For example, there can be certain design requirements and components that influence both the engine and the interior of a car. The shape of the entire body might in turn depend on the size of the engine and the space in the interior besides other factors. There can be different groups of people assigned to working on different parts of the car. For instance,

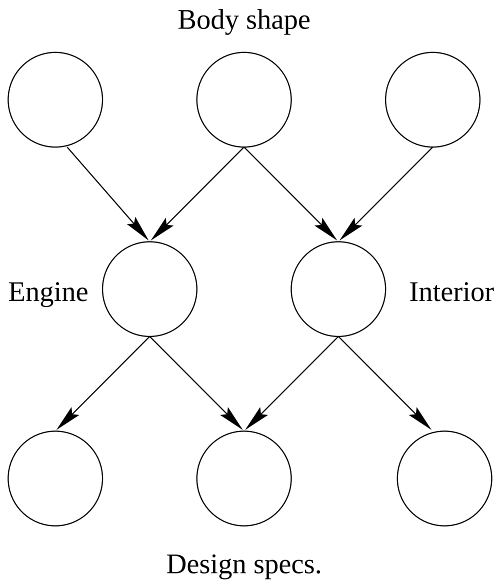

Figure 6.1: Concurrency control

one of the groups (say group A) may be assigned to work on designing the engine, while the other group (group B) could be working on the interior design of the car including seating and comfort. In such a design scenario, if the traditional concurrency control scheme of two-phase locking is employed, group A needs to lock the objects involved in the design of the engine. Since any modifications in the parts that influence the engine would change the design of the engine, it should also lock all the objects involved in the prerequisites of the engine components (design specs, in Figure 6.1. This itself would prevent group B from proceeding with their design since they need to lock the prerequisites of their components that were already locked by group A. Thus all the concurrency in the design process is lost. Besides, the standard concurrency control strategies are not appropriate for design scenarios in CAD applications for several other reasons [17]:

Long Transactions: Operations on objects in design environments are often long-lived. Blocking all the resources until a transaction commits would result in substantial reduction in concurrency and collaboration.

User Control: Unlike the sequence of actions in traditional database applications (such as banking and airline applications), the actions in graphics applications are more interactive in nature. The conventional transaction model does not support any user intervention within a transaction. The only way a transaction could be tested for consistency is by executing the transaction and checking the state of the database. If there is any consistency violation, the entire transaction has to be rolled back. With long transactions, this would mean that the user has to sacrifice all the amount of work. As opposed to rejecting the entire transaction, the user might want to make a set of changes to his design to bring the model back to the state where it does not violate the consistency requirements. Thus there is a need to provide more user control on the transaction.

Synergistic Cooperation: One of the most common facets of a design setting is the collaboration between the designers. In a collaborative environment, designers would like to work on shared objects, exchange objects, and even modify parts of the same object simultaneously. None of these levels in collaboration could be achieved by any serial schedule. Thus insisting on serializable concurrency control might prevent the desirable forms of cooperation between designers.

The underlying problem with existing concurrency control strategies is that they are all based on the strong notion of serializability. To ensure serializability, most of the concurrency control protocols abstract the application behavior into two classes of operations - *read an object* from the database, and *write an object* to the database. Without any further knowledge of the application semantics, all the applications can be made to follow the serializability constraint (and thus consistency), by preventing *read-write* and *write-write* conflicts. Thus, equating the notions of consistency with serializability causes a significant loss of concurrency in graphics applications. In the next section we propose some looser versions of concurrency control that help in maintaining consistency of data in a graphics application that is characterized by long transactions in addition to the requirements on user control and synergistic cooperation.

6.2 Concurrency Control Strategies

To design concurrency control strategies for CAD transactions, it would be helpful to have a good understanding of the development and design patterns that are common in these applications. Developers of a large project often divide themselves into small groups which work on specific components of the project. The interaction behavior between members of the same group will be different from the interaction between the groups. This results in a hierarchical organization of the entire project and reduces the problem of maintaining consistency into two sub problems: a flexible concurrency control scheme to allow cooperation between members of the same team, and a global policy that ensures correct serialization of the efforts of all the groups. This paradigm, which is referred to as the *group* paradigm in literature, is shown in Figure 6.2. Normally, each group will also be associated with a *local repository* to store the objects constructed or modified within the group, as opposed to a *global repository* which consists of all the finalized objects. Since there will be less overlap among the efforts of different groups, we can enforce a strict two-phase locking policy to control the concurrency at the top most level (global repository). The same policy will not be applicable between members of the same group for the reasons discussed in the previous section. The example in Figure 6.2 shows a two-level hierarchy. This could easily be extended to more than two levels. We first describe the two-phase locking at the top-most level, and then we look at schemes based on *version management* that support cooperation while maintaining consistency at the lower levels.

Referring to our example in Figure 6.1, the project of designing a car could be divided into two groups, one to handle the design of the engine, and the other to handle the interior design. We make a simplifying assumption that the complete shape of the car is determined by the designs of these two groups and some other factors which are constants. The global repository stores the entire model, whereas the local repositories are used to store the model for the engine and the model for the interior structure of the car. Most of the design updates will occur on the local repositories. Once a group decides that it has successfully finished a version of

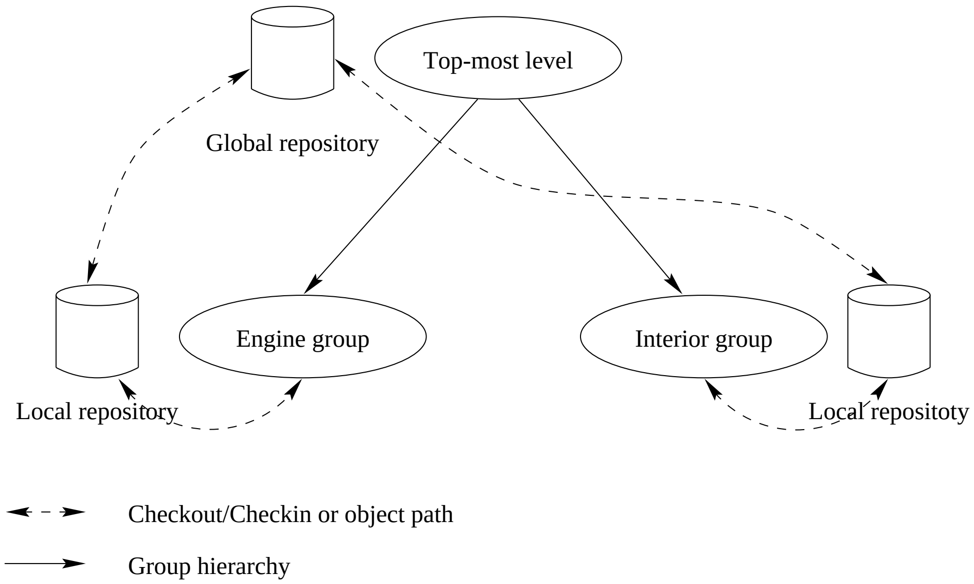

Figure 6.2: Group paradigm

the design, it can *checkin* the model into the global repository. Since this checkin process is not a long transaction, the group that needs to checkin is free to lock the model in the global repository for the duration of the checkin. The other groups in the project can see the work of a particular group only when it checks in its model into the global repository. This is not a concern since the groups were originally designed to work in isolation. Once the checkin process is complete, the dependencies can be propagated in the entire model resulting in an updated version of the car in the global repository.

The problem of concurrency control at the level of local repositories is non trivial. The concurrency control scheme should support long transactions, user control, and synergistic cooperation. We start with a simple version management system to maintain consistency in the local repositories. Then we discuss the drawbacks of this scheme and propose extensions to support the required features.

Version management [18] is the term used to describe the set of organizational concepts and operational mechanisms for arranging engineering design data into hierarchical aggregates that change over time. The basic principle in a version management system is to control access to the shared objects so that only one developer can modify an object at any time. A standard approach that is widely implemented by the version control tools such as the Source Code Control System (SCCS) and the Revision Control System (RCS) is the *checkout/checkin* mechanism. Under this approach, each data object is considered to be a collection of different versions. Each version represents the state of the object at some point of time in the history of its development. Once a version of an object is created, it becomes *immutable*, which means that it cannot be changed. Instead, a new version can be created after explicitly reserving the object. The reservation makes a copy of the original object and gives the owner of the copy exclusive rights on the copy so that he or she can modify it and deposit as a new version.

Two or more users can work on the same object only by creating multiple parallel versions of the object, creating branches in version history. In the presence of graphs of objects, a version of an object consists of all the objects that can be reached from that object in the graph. For the rest of the discussion, we extend our example of Figure 6.1 by expanding the engine of the car into a hypothetical model shown in Figure 6.3. When a designer X checks out object C from the engine model, the entire subgraph rooted at object C is checked out - in this case, objects C and D. Once these objects are modified, they can be checked in as new versions C_1 and D_1. Let us assume that another designer Y has decided to checkout object C while designer X is working on his or her objects. The modifications made by Y will be deposited as another parallel version D_2, once Y checks in his updates. The engine model in the local repository looks like the one shown in Figure 6.4 after the updates from X and Y. The trees within the ellipses show the version histories for objects C and D.

From the representation of Figure 6.4, it would not be possible be tell which versions of different objects are consistent with each other. Hence, there is a need to group sets of versions consistent with each other into *configurations*. This would enable designers to reconstruct a version of the entire model. Three valid configurations exist for the representation of Figure 6.4. They are shown in Figure 6.5.

Having described the basics of the version management system, we now describe

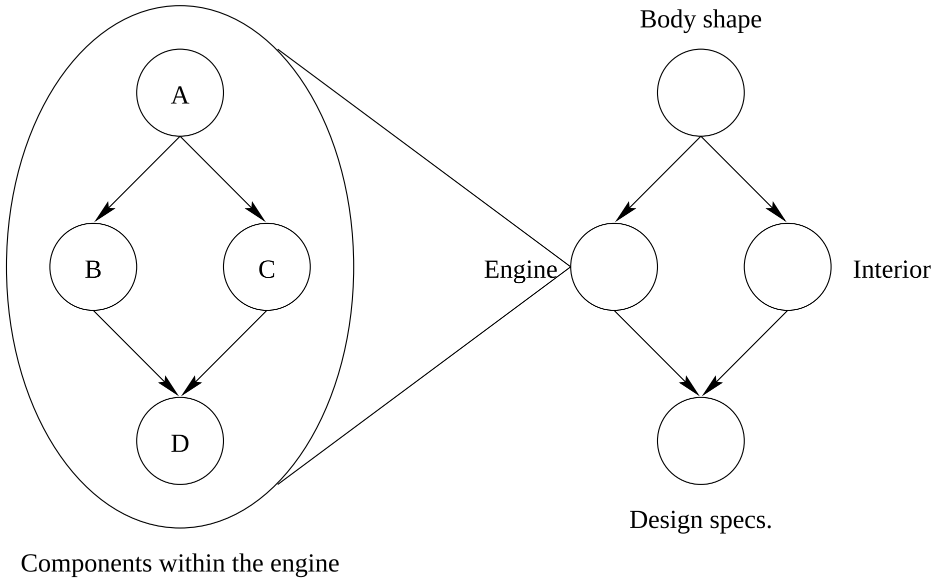

Figure 6.3: Subcomponents of the engine model

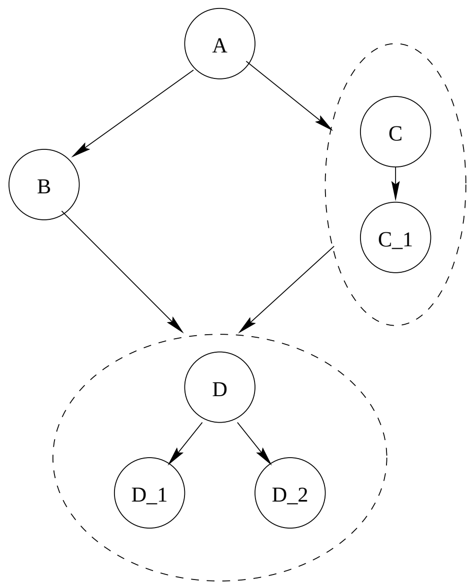

Figure 6.4: Version histories within the engine model

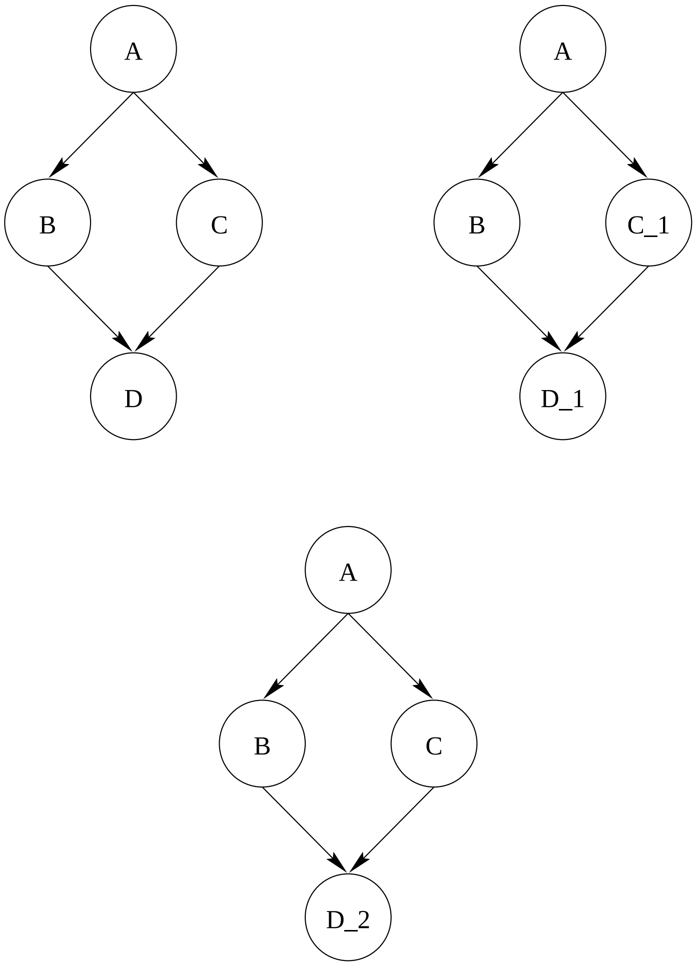

Figure 6.5: Configurations in version management

its effectiveness in supporting the features required for transactions in graphics applications.

6.2.1 Long Transactions

Designers who need access to the same object must either wait until the new version is deposited or reserve another version if that exists. Since the deposited versions are locked only for a limited amount of time (until they are copied into a new version), designers can almost always find a version of the object that can be checked out. Designers execute long transactions on their checked out copies before checking them back to the repository and depositing new versions. Thus, this model supports long transactions to some extent. However, when a designer needs to wait on a particular checked out version of an object, he has to wait until the object is deposited back into the repository. This would decrease concurrency and collaboration, and is discussed in detail in section 6.2.3.

6.2.2 User Control

No rollbacks are ever required in this model, since updates to objects are always deposited as new versions. However, user control is still required when two versions of the same object have to be *merged*. To demonstrate this, consider the example in Figure 6.4. After the updates by designers X and Y are checked in, they might decide that the final version of the engine model should contain part of the updates made by X, and part of the updates made by Y. So, objects D_1 and D_2 have to be merged with each other accordingly. The process of merging can be facilitated by a *merge editor* that displays both the objects D_1 and D_2 visually (if they can be displayed) or textually (by dumping the state of the object). One could then modify or create another object with the necessary state and deposit it back into the repository. The engine model after the merging process is shown Figure 6.6. Version management can thus be easily extended to support more user control.

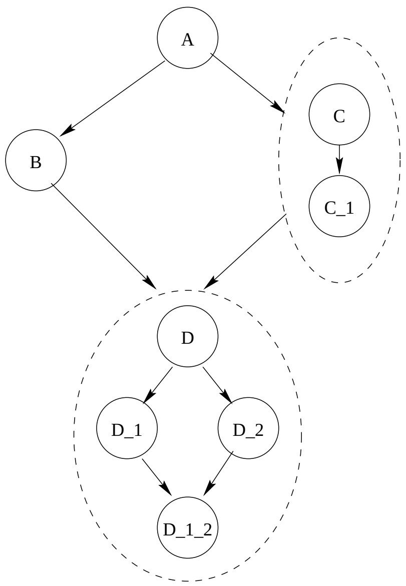

Figure 6.6: Merging of different versions

6.2.3 Synergistic Cooperation

The main drawback with the version management system is its lack of support for collaboration among different designers. Designers can work on only parallel versions of the model at a time. However, we can extend the version management system with a set of lock modes that could be used to improve cooperation among different designers. The extended model supports three different lock modes on a version of an object: (a) read only, which makes a version available only for reading, (b) shared derivation, which allows the owner to both read the version and derive a new version from it, while allowing parallel reads of the same version and derivation of different new versions by other users, and (d) exclusive lock, which allows the owner to read, modify, and derive a version and allows no parallel operations on the locked version.

The exclusive locks can be used to isolate the efforts of independent developers (as in the top-most level of the project hierarchy). If a designer requests a lock on an object that is already locked in an incompatible mode, the lock is rejected and initiator is informed of the rejection. This prevents deadlocks, which are caused by blocking of transactions that wait for unavailable resources. The initiator could be informed when the requested lock is available. Further, all the locks have to be acquired before any of them is released. However, an acquired lock can be strengthened to a more powerful lock in the lock acquiring phase, and weakened in the lock releasing phase. The read only lock is compatible with any lock mode, and this lets a user to watch the progress of another designer without affecting the designer's work. To be able to work on the same object, locks can be requested in the shared derivation mode. To explain this, we return to our example in Figure 6.3 where designers X and Y need to modify object D simultaneously. X requests a shared derivation lock on D, and Y requests a shared derivation lock on the version checked out by X. Designer Y can now watch all the updates done by X on D_1. At the same time, Y can derive a new version of object D, called D_2, and perform his updates accordingly on D_2. Since, Y is already aware of the updates as and when they are made by X, the process of merging D_1 and D_2 will be either trivial or very simple.

The mechanism described here supports collaboration of work to a small extent. Synergistic cooperation where users can freely exchange objects, or where many users can simultaneously work on the same object, remains unsolved.

CHAPTER 7

ALPHA_1 APPLICATIONS

So far, we have discussed the features of our object migration system and explained how we have implemented it. In this chapter, we demonstrate the feasibility and ease of using the system to incorporate object migration in existing applications. The experiments in this chapter also help us to empirically demonstrate the functional correctness of our system.

Alpha_1 is an integrated graphics, modeling, design and manufacturing package based on B-splines. The Alpha_1 group is actively engaged in fundamental and applied research in developing methods for representing, specifying, manipulating, and visualizing geometric models, including assemblies and mechanisms, on a computer, as well as associated process planning, and manufacturing issues for both traditional (milling, turning, CMM, EDM), and innovative (layer technology) manufacturing processes. The rest of this chapter describes some of the existing facets of Alpha_1, and how they can be modified to benefit from the advances in distributed systems and object migration.

Alpha_1, as it exists today, is comprised of numerous independent applications such as a renderer, a model constructor (`c-shape-edit`), and a model viewer (`motif3d`). All these applications themselves use object-oriented design and are implemented in C++. The `c-shape-edit` is the central core of the entire Alpha_1 system which provides a programming language interface for constructing models of any complexity. It supports many operations for constructing different kinds of objects ranging from points, polygons, curves, and surfaces to assemblies and features such as pockets and holes. All these objects are arranged in an inheritance hierarchy to allow reuse and polymorphic behavior. A part of the current inheritance hierarchy for the fine-grain objects in Alpha_1 in shown in Figure 7.1. All the

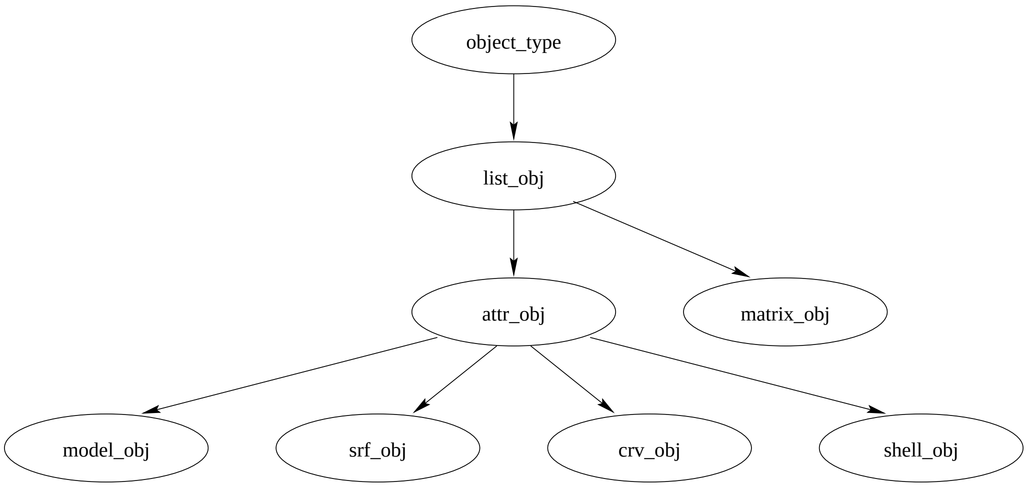

Figure 7.1: Hierarchy of objects in Alpha_1

objects inherit from a common `object_type`. Examples of these classes are given later in this chapter.

The independent applications of Alpha_1 interact by saving their output to a file where other programs can access it. For example, the model constructor could save the model to a `.a1` file, which can then be used by the renderer to render the model. Nicholas Rahn has provided object-oriented wrappers to these independent applications, so that each of them can make member function invocations on the other objects [19]. In this setting, the large-grain objects in Alpha_1 such as the model constructor, model viewer, and renderer will be running as CORBA servers, and any object is free to invoke methods on these objects. In the next section we describe this model in a greater detail, and explain how these CORBA objects can be extended to support object migration using our system of object migration. Then, we describe some of our attempts in converting the huge collection of fine-grain objects into objects that can freely migrate between applications.

7.1 Large-grain Objects in Alpha_1

There are distributed CORBA implementations for four large-grain objects in Alpha_1 - the model repository, the renderer, the model viewer, and the model constructor. The interfaces for a renderer object and a model repository object are shown in Figure 7.2. The renderer object supports one method which accepts

```
interface renderer
{
   oneway void render_to_file( in a1_object a1obj,
                               in string file_name );
};

interface model_rep
{
   a1_object query( in string model_name );
   void add_model( in string model_name, in a1_object a1obj );
   void remove( in string model_name );
   a1_object read_file( in string file_name );
};
```

Figure 7.2: Large-grain objects in Alpha_1

an Alpha_1 model and renders the model to a specified file. The model repository supports methods for storing, retrieving, and removing Alpha_1 models in or from a repository. The objects can be running as CORBA servers which accept requests from any Alpha_1 application. We have extended the current Alpha_1 system by implementing object migration for these large-grain objects. In addition to providing an object-oriented interface, these large-grain objects can now migrate on the network between nodes, which results in all the benefits mentioned in Chapter 2.

The methodology for adding migration capabilities to these objects is exactly similar to that described in Chapter 4. Each of the interfaces is made to inherit from `CosLifeCycle::LifeCycleObject` and `CosStream::Streamable`, as shown in Figure 7.3. The implementation for these objects should then provide the definitions for all the methods required to externalize or internalize an object, and the lifecycle operations including move, copy, and replicate. The tools in our object migration system make the task of providing these implementations very simple. A complier called `lggen` takes the interfaces (Figure 7.3), and generates the code for all the lifecycle operations. Once the data members in the implementation class have been specified, the same compiler does a second pass on the implementation

```
interface renderer : CosLifeCycle::LifeCycleObject,
                     CosStream::Streamable
{
   oneway void render_to_file( in a1_object a1obj,
                               in string file_name );
};
```

Figure 7.3: Large-grain objects with lifecycle operations

class declaration, and generates the code for the externalization and internalization operations. A simplified version of the implementation for the renderer object, and the code that is generated by our compiler is shown in Figure 7.4.

7.2 Fine-grain Objects in Alpha_1

Alpha_1 supports nearly 500 object types that are used in the modeling and design of various geometric and mechanical parts. To demonstrate the usability of our object migration system, we have provided object migration support to a small subset of these object types. Two of the types that are widely used in Alpha_1 are the surface object type (`srf_obj`) and the curve object type (`crv_obj`). These object types were extended to support migration capabilities. A simplified version of the class declaration for `srf_obj` is shown in Figure 7.5. Notice that the data members in a surface object include matrix objects (for storing the control mesh and knot vectors of the surface), shell objects (a pointer to the parent shell that contains this surface), and other objects such as the trimming objects and tear objects (which are used in characterizing certain special properties of the surface). All these objects should also be made to support object migration. Further, the surface object type inherits from an attribute object type (`attr_obj`) which inherits from list object type (`list_obj`), which in turn inherits from (`object_type`) as shown in Figure 7.1. This implies that all the base class objects should also be made to support object migration.

The methodology for adding migration support to these fine-grain objects is also exactly similar to the one mentioned in Chapters 4 and 5. The base class

```
class _im_render :
   public _sk_render,
   public _sk_CosLifeCycle::_sk_LifeCycleObject,
   public _sk_CosStream::_sk_Streamable
{
private:
   // private data members...

public:
   // Lifecycle operations...
   CosLifeCycle::LifeCycleObject_ptr copy(
      CosLifeCycle::FactoryFinder_ptr there,
      const CosLifeCycle::Criteria& the_criteria);
   void remove();
   void move(CosLifeCycle::FactoryFinder_ptr there,
             const CosLifeCycle::Criteria& the_criteria);
   CosLifeCycle::LifeCycleObject_ptr replicate(
      CosLifeCycle::FactoryFinder_ptr there,
      const CosLifeCycle::Criteria& the_criteria);

   // Externalization and Internalization operations...
   void internalize_from_stream(
      CosStream::Stream_ptr sourceStreamIO,
      CosLifeCycle::FactoryFinder_ptr there);
   void externalize_to_stream(
      CosStream::Stream_ptr targetStreamIO);
};
```

Figure 7.4: Code generated by `lggen`

```
typedef matrix_obj mat_1d_obj;
typedef matrix_obj mat_2d_obj;

class srf_obj : public attr_obj
{
    // private data members
    trimming_loop_obj *trim_;
    mat_3d_obj *s_mesh_;
    tear_info_obj *t_info_;
    shell_obj *shell_;
    mat_1d_obj *s_kv_[2];
    int s_order_[2];
    ec_type s_type_[2];
public:
    // member functions...
};
```

Figure 7.5: Surface object type in Alpha_1

of the hierarchy is made to inherit from `fgObject`, and the externalization and internalization operations are defined for each object type. This process is simplified by one of our tools called `fggen`. `fggen` is a compiler which takes a normal C++ class declaration and converts all the pointers in it systematically into smart pointers. It also automatically generates the code for externalizing and internalizing objects of that type, while considering all the issues mentioned in Chapter 6 such as object sharing within the graph and migration policies.

In order to test our object migration system, we have implemented simple Alpha_1 applications which generate some curves and surfaces. One of these applications is shown in Figure 7.6. The generated surfaces (or curves) are then migrated into another application shown in Figure 7.7. The surface objects are then compared by dumping them to Alpha_1 streams and viewing them using model viewers.

```
#include <alpha_1.h>

int main( int argc, char **argv )
{
   // initialization
   CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
   CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);
   thread_t tid;
   fg_handler = init_callback_thread(boa, &tid, &mutex);
   if (!tid)
      exit(1);

   // create a simple surface
   pt_obj pobj;
   vec_obj vobj(1, 2, 3);
   crv_obj *cobj = create_unit_circle();
   srf_obj *sobj = srf_of_revolution( pobj, vobj, cobj );
   sobj->publish(``SRFOBJ'');
   sobj->dp_obj();

   // wait
   getchar();
}
```

Figure 7.6: Creation of a surface

```
#include <alpha_1.h>

int main( int argc, char **argv )
{
   // initialization
   CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
   CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);
   thread_t tid;
   fg_handler = init_callback_thread(boa, &tid, &mutex);
   if (!tid)
      exit(1);

   // migrate the surface
   representative_ptr r = srf_obj::subscribe(``SRFOBJ'');
   srf_obj *sobj = srf_obj::_narrow(r->move());
   sobj->dp_obj();
}
```

Figure 7.7: Migration of the surface object

CHAPTER 8

CONCLUSIONS

Having identified distributed computing as the computing paradigm of the future, we have discussed how distributed object-oriented platforms such as CORBA help in abstracting the complexities of distributing an application from the application semantics. CORBA has its foundation in many earlier distributed systems and platforms such as OSF DCE. The founders of CORBA were successful in compiling all the features necessary to create distributed applications, in the form of object services and common facilities.

We noticed that graphics applications cannot benefit directly from these advances due to several factors. The primary requirement for graphics applications, which was not well supported by CORBA, was the object model. Graphics applications demand an environment where fine-grain objects (which normally reside in the address space of the application that creates them) can migrate freely between applications. In order to meet this requirement we have added the support for fine-grain objects in CORBA by two additional object services in the form of representative objects and callback objects. With this addition, objects can now interact in a fine-grain manner within an application while still supporting their use by remote applications. Remote applications can migrate the fine-grain objects to their own address space before using them. This model requires the creation of one representative object (which is a CORBA object) for every fine-grain object created. Though this would mean that there will be a large number of representative objects, we have provided scalability in the creation of representative objects through representative object factories. A certain number of representative objects are created within a factory, and once the capacity of a factory is exceeded, new factories are started. Further, these factories could be started anywhere on the network.

We have identified the necessity for different degrees of migration in a collaborative environment. When users want to start working on a new version of an object, they can use the `copy` operation. When they need to work on the same object they can use the `replicate` operation. The `move` operation is used to completely transfer objects from one application to another. For example, it can be used to finally checkin the object into a repository. These different degrees of migration are also required for the concurrency control schemes discussed in one of the chapters. The three degrees of migration, together, provide a powerful set of primitives that can be used to effectively manage collaboration in design.

Since objects do not exist in isolation we have provided the support for migrating object graphs of arbitrary complexity. The common problems associated with migrating object graphs, such as cycles within a graph and sharing of objects in a graph, have been handled. To provide more control on the granularity of migration within a graph, we have discussed several migration policies including the shallow and deep migration policies, class boundaries, and diffusion model. The diffusion model, being the most powerful of the discussed models, can be used to simulate the other migration policies. With this granularity control, users can set the exact behavior of migration before invoking a migration operation on a graph of objects. In most of the cases, the right migration policy (which determines the performance) depends on the specific application semantics, while in others it can only be determined by repeated experiments.

To implement the migration policies and to solve certain language dependent features in abstract data types, we have introduced the notion of smart pointers. Smart pointers provided a very intuitive and elegant solution to all the problems associated with graphs of objects.

Concurrency control plays a very important role in the presence of replication of objects and object graphs. We have discussed the drawbacks of some of the standard concurrency control schemes when applied to graphics applications, and identified that any concurrency control scheme for a graphics application should support long transactions, more user control, and synergistic cooperation. Then, we proposed some techniques that are aimed at supporting these three features. The drawbacks of the techniques were also discussed.

To substantiate our results, we have tried out some experiments on the Alpha_1 system. Though it would be impractical to convert all the existing object types in the Alpha_1 system into migratable objects, we have identified a small subset of objects that would clearly demonstrate the feasibility of object migration in the system. Further, the existing implementation of the Alpha_1 system is not distributed in nature, partly for the reasons mentioned in Chapter 1, and partly for the reason that it has evolved over the past 20 years when there was not much interest in distributed systems. A major direction of growth in the current Alpha_1 system would be to make it more distributed so that there could be more support for collaboration in design.

Though our object migration system supports migration of fine-grain objects and object graphs, we have not provided any support for passing objects by value in remote method invocations. All the objects are passed by reference in CORBA, and passing objects by value has some special benefits in terms of performance. To pass an object by value, the object has to be migrated to the process where the remote method is executed, and then migrated back to the caller. Thus *call-by-value* can be supported as an extension to object migration. Further, other aspects of distributed object management such as object persistence could also be implemented as extensions to object migration. Object migration also opens up new areas of research in collaborative design environments. One could design tools which are very helpful for supporting collaboration, by taking advantage of object migration.

APPENDIX

IDL FOR OBJECT SERVICES

```
/* Naming Service */

module CosNaming
{
   typedef string Istring;
   struct NameComponent {
      Istring id;
      Istring kind;
   };
   typedef sequence <NameComponent> Name;
   enum BindingType {nobject, ncontext};
   struct Binding {
      Name binding_name;
      BindingType binding_type;
   };
   typedef sequence <Binding> BindingList;

   interface NamingContext {

      enum NotFoundReason { missing_node, not_context, not_object};
      exception NotFound {
         NotFoundReason why;
         Name rest_of_name;
      };
      exception CannotProceed {
         NamingContext cxt;
         Name rest_of_name;
      };
      exception InvalidName{};
      exception AlreadyBound {};
      exception NotEmpty{};

      void bind(in Name n, in Object obj)
         raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
      void rebind(in Name n, in Object obj)
         raises(NotFound, CannotProceed, InvalidName);
      void bind_context(in Name n, in NamingContext nc)
         raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
      void rebind_context(in Name n, in NamingContext nc)
         raises(NotFound, CannotProceed, InvalidName);
      Object resolve (in Name n)
         raises(NotFound, CannotProceed, InvalidName);
```

```
      void unbind(in Name n)
         raises(NotFound, CannotProceed, InvalidName);
      NamingContext new_context();
      NamingContext bind_new_context(in Name n)
         raises(NotFound, AlreadyBound, CannotProceed, InvalidName);
      oneway void remove( )
         raises(NotEmpty);
      BindingList list();
   };
};

/* LifeCycle Service */

module CosLifeCycle{

   typedef CosNaming::Name Key;
   typedef Object Factory;
   typedef sequence <Factory> Factories;
   typedef struct NVP {
      CosNaming::Istring  name;
      any     value;
   } NameValuePair;
   typedef sequence <NameValuePair> Criteria;
   enum add_replica_mode {recursive, not_recursive};

   exception NoFactory {
      Key search_key;
   };
   exception NotCopyable { string reason; };
   exception NotMovable { string reason; };
   exception NotRemovable { string reason; };
   exception NotReplicable { string reason; };
   exception InvalidCriteria{
      Criteria invalid_criteria;
   };
   exception CannotMeetCriteria {
      Criteria unmet_criteria;
   };


   interface FactoryFinder {
      Factories find_factories(in Key factory_key)
         raises(NoFactory);
   };

   interface LifeCycleObject {
      LifeCycleObject copy(in FactoryFinder there,
                           in Criteria the_criteria)
         raises(NoFactory, NotCopyable, InvalidCriteria,
                CannotMeetCriteria);
      void move(in FactoryFinder there,
                in Criteria the_criteria)
         raises(NoFactory, NotMovable, InvalidCriteria,
                CannotMeetCriteria);
```

```
      oneway void remove()
         raises(NotRemovable);

      // the following operations are provided for replication
      LifeCycleObject replicate(in FactoryFinder there,
                                in Criteria the_criteria)
         raises(NoFactory, NotReplicable, InvalidCriteria,
                CannotMeetCriteria);
      void add_replica(in LifeCycleObject obj,
                       in add_replica_mode mode);
      void sync_in(in LifeCycleObject obj);
      void sync_out();
   };
};

/* FactoryFinder based on Naming Contexts */

interface NC_FactoryFinder :
CosNaming::NamingContext, CosLifeCycle::FactoryFinder {

   /*
    * This is a factory finder that is based on naming contexts
    * in naming service. See page-72 of COSS-1 for more details.
    */
};

/* Externalization Service */

module CosStream {

   exception ObjectCreationError{};
   exception StreamDataFormatError{};

   interface Streamable;

   interface Stream {
      void externalize(
         in Streamable theObject);
      Streamable internalize(
         in Streamable theObject)
         raises( StreamDataFormatError );

      void flush();

      void write_string(in string aString);
      void write_char(in char aChar);
      void write_octet(in octet anOctet);
      void write_unsigned_long(
         in unsigned long anUnsignedLong);
      void write_unsigned_short(
         in unsigned short anUnsignedShort);
      void write_long(in long aLong);
      void write_short(in short aShort);
      void write_float(in float aFloat);
```

```
      void write_double(in double aDouble);
      void write_boolean(in boolean aBoolean);
      void write_object(in Object obj);

      string read_string()
         raises(StreamDataFormatError);
      char read_char()
         raises(StreamDataFormatError );
      octet read_octet()
         raises(StreamDataFormatError );
      unsigned long read_unsigned_long()
         raises(StreamDataFormatError );
      unsigned short read_unsigned_short()
         raises(StreamDataFormatError );
      long read_long()
         raises(StreamDataFormatError );
      short read_short()
         raises(StreamDataFormatError );
      float read_float()
         raises(StreamDataFormatError );
      double read_double()
         raises(StreamDataFormatError );
      boolean read_boolean()
         raises(StreamDataFormatError );
      Object read_object()
         raises(StreamDataFormatError );

      oneway void remove();
   };

   interface StreamFactory {
      Stream create_object();
   };

   interface Streamable {
      void externalize_to_stream(
         in Stream targetStreamIO);
      void internalize_from_stream(
         in Stream sourceStreamIO,
         in CosLifeCycle::FactoryFinder there)
         raises( CosLifeCycle::NoFactory,
                 ObjectCreationError,
                 StreamDataFormatError );
   };
};

/* Representatives for Fine-grain objects */

module fine-grain {

   interface representative {
      attribute CosStream::Stream my_stream;
      attribute callback event_handler;
```

```
      representative copy(inout Traversal t)
         raises ( CosLifeCycle::NotCopyable );
      void move(inout Traversal t)
         raises ( CosLifeCycle::NotMovable );
      representative replicate(inout Traversal t)
         raises ( CosLifeCycle::NotReplicable );
      void remove()
         raises ( CosLifeCycle::NotRemovable );

      void add_replica(in representative fg_obj,
                       in CosLifeCycle::add_replica_mode mode)
         raises ( CosLifeCycle::NotReplicable );
      void sync_in(in representative fg_obj);
      void sync_out();
   };

   interface representativeFactory {
      exception CapacityExceeded{ };

      representative create_object()
         raises ( CapacityExceeded );
   };
};

/* Callback Objects */

interface callback {
   void send_message(in string obj_name, in string op_name,
                     inout CosStream::Stream s,
                     inout Traversal t);
};
```

REFERENCES

[1] A.S. Tanenbaum, *Distributed Operating Systems.* Upper Saddle River, New Jersey 07458: Prentice Hall, 1995.

[2] M. Linton and C. Price, "Building Distributed User Interfaces with Fresco," in *Proc. Seventh X Technical Conference*, (Boston, Masachusetts), pp. 77–87, Jan. 1993.

[3] R.M. Soley and C.M. Stone, *The Object Management Architecture Guide.* John Wiley & Sons, Inc., third ed., July 1995.

[4] Object Management Group, "The Common Object Request Broker: Architecture and Specification (CORBA)," July 1995.

[5] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-grained Mobility in the Emerald system," *ACM Transactions on Computer Systems*, vol. 6, pp. 109–133, Feb. 1988.

[6] R.J. Fowler, *Decentralized Object Finding Using Forwarding Addresses.* PhD thesis, University of Washington, Seattle, Dec. 1985.

[7] P.Y. Chevalier, D. Hagimont, J. Mossiere, and X.R. DePine, "Object Migration in the Guide System," Tech. Rep. 101, Broadcast, 1995.

[8] S.J. Caughey, G.D. Parrington, and S.K. Shrivastava, "SHADOWS - A Flexible Support for Objects in Distributed Systems," in *IWOOOS'93*, Dec. 1993.

[9] R.S. Chin and S.T. Chanson, "Distributed Object-based Programming Systems," *ACM Computing Surveys*, vol. 23, pp. 91–124, Mar. 1991.

[10] A.S. Tanenbaum and R. van Renesse, "Distributed Operating Systems," *ACM Computing Surveys*, vol. 17, pp. 419–470, Dec. 1985.

[11] M. Rozier and J.L. Martins, *Distributed Operating Systems, Theory and Practice*, ch. The CHORUS Distributed Operating System; Some Design Issues, pp. 262–287. Berlin, Heidelberg: Springer-Verlag, 1987.

[12] P. Dasgupta, R. LeBlanc, and W. Appelbe, "The Clouds Distributed Operating System - Functional Description, Implementation Details, and Related Work," in *IEEE 8th International Conference on Distributed Computing Systems*, (San Jose), 1989.

[13] Object Management Group, "Common Object Services Specification, Volume

I," tech. rep., Object Management Group, 1994.

[14] International Business Machines Corporation and SunSoft, Inc, "Object Externalization Service," tech. rep., Object Management Group, 1994.

[15] B. Stroustrup, *The C++ Programming Language.* Addison-Wesley Publishing Company, 1991.

[16] H. Korth and A. Silberschatz, *Database System Concepts.* McGraw-Hill, 1986.

[17] N.S. Barghouti and G.E. Kaiser, "Concurrency Control in Advanced Database Applications," *ACM Computing Surveys*, vol. 23, pp. 269–317, Sept. 1991.

[18] R.H. Katz, "Toward a Unified Framework for Version Modeling in Engineering Databases," *ACM Computing Surveys*, vol. 22, pp. 375–408, Dec. 1990.

[19] N. Rahn, "BORG: A System for Assimilating Legacy Systems into Distributed Objects," Master's thesis, University of Utah, 1996.