Context-Sensitive Multimedia

by

Nathan Shalom Abramson

S.B., Computer Science and Engineering
Massachusetts Institute of Technology
(1990)

SUBMITTED TO THE
MEDIA ARTS AND SCIENCES SECTION, SCHOOL OF ARCHITECTURE
AND PLANNING
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE
DEGREE OF
MASTER OF SCIENCE IN VISUAL STUDIES

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February 1993

Signature of Author: [illegible]
Media Arts and Sciences Section
September 16, 1992

Certified by: [illegible]
Walter Bender
Principal Research Scientist, MIT Media Lab
Thesis Supervisor

Accepted by: [illegible]
Stephen A. Benton
Chairperson
Departmental Committee on Graduate Students

Context-Sensitive Multimedia

by

Nathan Shalom Abramson

Submitted to the Media Arts and Sciences Section, School of Architecture and Planning
on September 16, 1992 in partial fulfillment of the
requirements for the Degree of
Master of Science in Visual Studies at MIT

ABSTRACT

The current growth of multimedia applications illustrates the need for standardized object libraries. This thesis suggests that an object-oriented context-sensitive approach may provide the needed flexibility. This thesis describes the construction of an environment for developing and experimenting with context-sensitive multimedia objects. These objects are designed for use across a wide variety of applications: they are active processes which can adapt themselves to the context in which they are used.

The research testbed for object-oriented multimedia developed in this thesis is called O. The design and implementation of O is described. The initial use of O to support context-sensitive multimedia is examined in two applications: an interactive movie map and a personalized news presentation.

Thesis Advisor: Walter Bender
Principal Research Scientist, MIT Media Lab
This work was supported in part by International Business Machines Incorporated.

Context-Sensitive Multimedia

by

Nathan Shalom Abramson

Thesis readers

Reader: [illegible]

Glorianna Davenport
Assistant Professor of Media Technology, MIT Media Arts & Sciences Section

Reader: [illegible]

Tod Machover
Professor of Music and Media, MIT Media Arts & Sciences Section

Contents

1 Introduction **1**

1.1 Promises 1

1.2 Paradigms 3

1.3 Adaptability 4

1.4 Context-sensitive multimedia 5

1.5 Context-sensitive multimedia and current multimedia systems 7

1.6 O 8

2 O **10**

2.1 Glossary 11

2.2 Story of O 15

2.3 Module networks 24

2.4 OSpace 42

2.5 Transportable Objects 49

2.6 The Host/OStream model 55

2.7 Applications Interface 60

2.8 Future directions for O 62

3 Context-Sensitive Multimedia **63**

3.1 Object-Oriented Multimedia 63

3.2 Context-sensitive multimedia 72

3.3 Applications of context-sensitive multimedia 73

3.4 Context 74

3.5 Contextual hierarchy 78

3.6 Context-sensitive multimedia and O 79

4 Demonstrations **85**

4.1 The Bush Demo 85

4.2 The Interactive Hallways Demo 88

5 Conclusion **93**

6 Acknowledgments **97**

List of Figures

2.1 Movie player with many decoders 16

2.2 Universal movie player using embedded decoders 17

2.3 Levels of programming in O 20

2.4 Module network split across machines 23

2.5 A module network 25

2.6 An OModule 27

2.7 A module group 29

2.8 A remote alias module 30

2.9 A module group inheriting configuration information 34

2.10 Immediate versus queued message delivery 38

2.11 An OStream 40

2.12 Remote object location using Oservers 45

2.13 Oserver starting an OStream 45

2.14 Stdio OModule class type 47

2.15 Dtype client OModule class type 48

2.16 OModules being transported 51

2.17 OStream forking 55

2.18 Host/OStream model 58

2.19 Application interface 60

3.1 Context-sensitive movie object 81

4.1 Object-oriented State of the Union 86
4.2 The Bush demo system 87
4.3 Context-sensitive hallways of MIT 90
4.4 O schematic for the interactive hallways demo 91

Chapter 1

Introduction

1.1 Promises

Multimedia is the term used to describe presentations that involve multiple modes of communication. Traditionally these modes have included film, video, photos, drawings, sound, and text. These media forms become more effective, memorable, and entertaining when used in combinations.

However, the process of combining multiple media elements is difficult. Creating multimedia presentations requires knowledge of design techniques, as well as a working knowledge of several media forms. It is also expensive, requiring equipment that is specialized to each media type (slide projectors, video machines). Once a multimedia presentation is created, the actual display of the work may once again require a great deal of knowledge and equipment. Finally, multimedia is extremely limited in distribution, e.g. a slide/video show is difficult to package and sell to thousands of customers.

For these reasons, the computer has become very popular as a multimedia creation/presentation device. A computer with the ability to manipulate media objects eliminates the need for individual pieces of expensive equipment dedicated to a single media form. Computers can also be multimedia display devices, so that the presentation of a multimedia work can be done with the same equipment that was used to create that work. Finally, computer technology has made great advances in the ability to distribute information between people. Multimedia can take advantage of this technology, resulting in multimedia presentations that can be packaged and distributed.

Recent technological advances have brought computer, or electronic, multimedia within affordable reach of the population. The result has been an explosion of research in multimedia technology, as well as countless methods and approaches for actually using multimedia in an electronic form. The first phase of the explosion is over, meaning that just about all traditional media forms now have electronic representations (however crude they may be) that can be manipulated by a computer. Photos, video, and film are now digital video. Sound is now digital audio. Printed text in all its richness and variation can also be represented, in one of many electronic forms.

The next step is to figure out how to put these multimedia elements together. Vendors, programmers, and designers have been working on this problem, each one crafting a unique way of thinking about electronic media integration, and a unique approach for carrying it out. What is needed now is a consolidation of these results, with the goal of a common model of multimedia integration.

While this common model will have its roots in traditional non-electronic multimedia, there will be major differences. Computers became popular in multimedia because they made multimedia cheap and easy to use. But computers have far more potential because they are "intelligent": they know about their users, they know about their environment, and they know about other computers they can talk to. A computer can take a multimedia presentation one step farther than the multimedia designer could by personalizing the presentation, enriching the presentation, and combining the presentation with other presentations.

1.2 Paradigms

The most popular approach for integrating multimedia is to create an *application* which uses multimedia *objects*. The application is in complete control of the presentation; the objects provide data for the application to use. In such an arrangement, the application and its data tend to be *tightly coupled*, which means that the multimedia objects will only work on that particular application. However, some of those objects may be valuable in other applications, so there should be some way of decoupling the applications from their objects. For this reason, several *interchange standards* are evolving which provide a common way of representing multimedia data. This will allow multimedia designers to create presentations that can be used in several applications. "Libraries" of shared multimedia objects can be created, which designers can access to create presentations (assuming copyright laws do not interfere). And multimedia can also be transported using these standards - the MIME standard [BF92], for example, is a method for transporting multimedia through Internet mail.

As multimedia objects begin to proliferate in this manner, two things will happen. First, multimedia objects will become more complex, with more interactive behavior. For instance, an object may display itself in a more text-oriented or a more video-oriented manner, depending on what the user prefers. Second, multimedia applications will become more dynamic, with presentations being assembled out of objects received "on the fly". For example, a multimedia news reader will constantly be updating itself with new multimedia news items received from news servers.

These changes will require applications to give up their total control over the presentation. The application will no longer know all about the multimedia objects it will have to handle, nor will the application be able to interpret all the behaviors a multimedia object may need to carry out.

As control shifts away from the application, that control is going to have be taken up somewhere else. The logical choice is to give that control over to the multimedia objects. But not all control should be shifted to the objects. Instead, the behavior of the system will be shared between application and objects. The application knows the direction of its presentation and what the user wants to see. The objects know their content and how they would best be presented in different situations. Together, the application and the objects can "negotiate" and work together to form the best possible presentation for the user.

1.3 Adaptability

The idea of application/object cooperation is going to require fundamental changes in the way that objects are represented. Currently, multimedia objects are strictly static data, usually encoded in one of several standard static formats. These objects may find themselves being used in various applications, on different platforms, for different users. It is unlikely that the same object will behave identically in all cases, especially if that object is a complete multimedia presentation. Therefore, the object must no longer be represented as static data, but as *adaptable* data. The following are examples of situations in which adaptable multimedia objects will be superior to static multimedia objects.

1.3.1 Scalable resource use

Displaying multimedia requires resources (e.g., screen space, network bandwidth, and computing resources), and some applications/platforms will have more resources than others. For example, a home entertainment center will probably have more bandwidth (and screen space) than a portable video player. Or, multiple media objects may be combined into a single application and will have to divide their resource usage according to their relative priority, as determined by the application. An adaptable multimedia object will be able to respond to varying availability of resources by scaling its operations accordingly. For example, an object may choose to reduce the size of its video to fit available screen space, or to switch to a text stream if the network bandwidth cannot support video data rates.

1.3.2 Presentation modes

Some multimedia objects will be able to present themselves effectively through a variety of presentation modes (e.g., video, audio, text). An adaptable object will be able to determine which modes to use, based on the display capabilities of the applications, the available resources, and the needs or preferences of the user. For example, the same multimedia news article may present itself in full video and text at the breakfast table, but switch to audio only when taken onto a crowded subway.

1.3.3 Content

An adaptable object will not only fit several platforms and displays, it will also fit several purposes. The same multimedia presentation may be called on in an instructional setting, or in an entertainment setting. A single presentation may be shown to different age groups or different nationalities, and will be expected to adjust itself accordingly. A multimedia news article might be placed into a conservative or a liberal newspaper, and be expected to rewrite itself to fit the tone of the newspaper. Multimedia with these abilities will be the valuable and versatile members of a shared multimedia library, and will also be the key to truly personalized applications.

1.4 Context-sensitive multimedia

If multimedia objects must be represented as adaptable data, how is this adaptability expressed?

The first natural approach is to express these multimedia objects in an *object-oriented* design. In this design method, an object comes complete with data and *methods*. To access the different behaviors of the object, an application would invoke the object's methods, thus causing the object to carry out its behavior according to the design built into the object. This way the application can use an object and the object's functions without knowing the inner workings and representations of the object.

Object-oriented design is a good first step towards making adaptable objects. With this design, the application can relate the current situation to the object by invoking certain methods, and the object can reconfigure itself as it sees fit.

Object-oriented multimedia is slowly appearing in commercial contexts. For example, *HyperMedia* [Ber90] has its own particular scheme for representing objects, a hypercard stack, which can also be used to represent multimedia. These hypercard stacks can easily be shared and reused by other programmers, which is one of the important qualities of object-oriented programming. *QuickTime* from Apple [Poo91] also takes a stab at an object-oriented approach for time-based media. QuickTime movie objects can easily be used by a wide variety of Macintosh applications. However, both of these systems do not completely espouse the object-oriented model since a great deal of the object-oriented design is hard-coded into the objects. Users are not completely free to add their own methods - in the case of HyperMedia, the scheme for linking objects is built into the design, and in the case of QuickTime, most of the runtime operation and interaction is hidden in the QuickTime objects. Despite these discrepencies, it is clear that the industry is starting to recognize object-oriented design as an important element in future multimedia applications.

However, object-oriented design still places the application in control of the situation. The object remains a passive entity which can only respond to directions from the application. As mentioned before, the object will have to take a more active role in the presentation for a number of reasons: the object may need to initiate a negotiation with the application for further resource allocation, the object may be in contact with other entities such as news servers, and will have to maintain those connections, etc. In other words, a multimedia object will be a self-aware, self-controlling, active process, which negotiates with, supplies data to, and receives user interaction from the application.

The concept of a multimedia object which is an active, self-aware entity is the central idea of *context-sensitive multimedia.* Context-sensitive multimedia is a model for representing active objects that are adaptable to a variety of situations, or *contexts.* Context-sensitive multimedia is also a model for building applications which establish and manipulate the contexts in which context-sensitive objects will operate.

1.5 Context-sensitive multimedia and current multimedia systems

As mentioned before, several current multimedia systems are beginning to espouse an object-oriented method in their design. Apple's QuickTime [Poo91] and Fluent's FM/1 [Flu90], for example, realize the benefits of encapsulating decoding behaviors, mostly as a means for gracefully extending applications to take advantage of evolving productions. In all cases, however, the application controls and keeps track of every piece of media that runs through the application. For example, a QuickTime movie contains several media tracks, each of which can hold video, sound, text - potentially a rich multimedia presentation. However, it is still the job of the application to figure out which tracks are appropriate to use and when, which is a lot to ask from the application. Even though the application may know what it is aiming for in terms of its presentation and content, the application is not necessarily familiar with the content of the movies it is playing. This means that the application is not in the best position to arrange an unknown multimedia object to its full potential.

Context-sensitive multimedia takes a different approach, by putting the application and the media on more equal footing. The application "knows" what it wants to present, the media "knows" what it can show. This "knowledge" might be represented as a series of complex behaviors, so it may be very difficult for the media to turn over all of its knowledge to the application, or vice versa. Instead, the application and the media keep their behaviors to themselves, but communicate with each other what they know about themselves. In effect, the media and the application negotiate with each other, towards the common goal of making the most effective and efficient use of the media's capabilities while still maintaining the direction and intent of the application.

This shift of decision-making from application to media is what is missing from current multimedia systems. The need for this shift will become more apparent as media objects become richer and multimedia research begins to focus on content selection and understanding.

1.6 O

O is a prototyping testbed for exploring the ideas of object-oriented and context-sensitive multimedia. O is itself a research project developed by the author over the past year. Being a research project, O includes as many different features as it can for exploring multimedia concepts. While this makes O somewhat large and bulky, it also allows O to be used to explore other topics besides object-oriented multimedia, such as networked digital video.

O is patterned after the object-oriented design method espoused by MAX [PZ] and Data Explorer [IBM91]. This method is based on the design of a real-time *process*, where the flow of data is directed between functional modules through a network of connections. Like MAX and Data Explorer, design and programming of an O data flow network is done through a graphical editor[1], rather than a text editor.

O is specially designed for object-oriented multimedia. Objects in O are *OModules*, which are "black boxes" with inputs and outputs. The inputs and outputs allow an OModule to be connected to other OModules or to an application. An OModule can be many things: a piece of code, an executable program, a server, or even a group

[1]The graphical editor for O, called Olga, was written by Marcel Bruchez and Eugene Lin.

of OModules. An OModule can be as complex or as simple as desired, which means that both high-level designs and low-level implementations can be expressed using the O model.

O's networked multimedia capabilities come from O's ability to distribute OModules to different machines on the network, while still keeping their connections intact. This distribution is kept abstract for the O programmer, which simplifies the mechanics of experimenting with networked multimedia systems.

Another of O's features is the ability to allow OModules to communicate with each other through an isolation barrier,. so that one faulty OModule can not crash the others. This feature is expanded later into the *Host/OStream* model, and is the model used to create applications that run context-sensitive multimedia.

The remainder of this thesis explores the techniques and applications of object-oriented/context-sensitive multimedia. The next chapter describes O in detail, and is aimed more towards those using this thesis as a programming reference for O. The remaining chapters explore the concepts of context-sensitive multimedia in theory and application. These latter chapters do not assume knowledge of O, but the concepts in these chapters will be more tangible if the reader is familiar with the concepts of O. Even if the reader completely skips the O chapter, the discussions in the remaining chapters should be understandable on an intuitive level.

Chapter 2

O

O is a research testbed for exploring issues in multimedia, specifically object-oriented, network-based multimedia. O was designed and developed by the author for the Entertainment and Information Technology group in the MIT Media Lab. O has been in development for over a year, and is currently in its third revision.

In addition to being a prototyping environment for object-oriented multimedia, O is also its own research project. The model for representing a distributed multimedia object has undergone several revisions throughout the development of O, a reflection of our changing ideas about object-oriented multimedia. O will continue to evolve, hopefully to incorporate the results of the multimedia research being performed with O.

The model for O is similar to the *ViewStation* project proposed by Chris Lindblad [Lin92], which is also based on a distributed object-oriented environment. ViewStation attempts to integrate the manipulation of temporally sensitive data with temporally insensitive processing by using *media entity* programming (a form of object-oriented programming), and *time-stamped streams* to move data around. However, the ViewStation project is primarily designed to meet temporally sensitive specifications, not high-level functional specifications. O is designed to implement high-level functional specifications, with the assumption that the O model can expand to include temporally sensitive functions.

This chapter outlines the motivation and evolution of O, followed by a detailed description of the O model. Consult [Abr92c] for a more complete description of O programming and implementation.

2.1 Glossary

The following is a glossary of the terms that will be used throughout this chapter.

Dtype Client OModule An OModule class generated by an OServer, which is handled by a Dtype server. Section 2.4.1.

Dtype environment A list of keyword/value pairs, used to allow OModules to inherit configuration information from parent OModules. Section 2.3.5.

Dtype reference A method of matching a value to a keyword in a Dtype environment, used to allow OModules to inherit configuration information from parent OModules. Section 2.3.5.

Dtypes A generalized data type that is easy to transport between machines. Dtypes are used to represent messages which are passed between O objects. Section 2.3.1.

Eval OModule A special OModule class which can be configured to a variety of behaviors through a SCHEME-like language. Section 2.3.5.

Executable OModule An OModule class generated by an Oserver, which is handled by a program that can have multiple inputs and outputs. Section 2.4.1.

Host/OStream isolation Part of the Host/OStream model, which guarantees that Hosts and OStreams are isolated, so that an object cannot easily crash an application. Section 2.6.1.

Host/OStream model A model for writing O applications, in which the application is a Host, which forks off OStreams that are used to run the objects which the application uses. Section 2.6.

O A system for creating and running distributed object-oriented processes. Section 2.

OModule The basic O object - a single functional unit with inputs and outputs. Section 2.3.2.

OModule class The type of the OModule. An OModule's behavior is determined by its class. Section 2.3.3.

OModule instance A specific OModule generated from an OModule class. Section 2.3.3.

OSCRIPT The original language for describing O objects - text-based, similar to C. Section 2.2.3.

OSpace The set of all OModules on all machines which can potentially connect with each other. Section 2.4.

OStream The actual program which runs a set of OModules, maintains a message queue, and also acts as a network server to communicate with other OStreams.. Section 2.3.8.

OStream forking The process of causing an OStream to split into two separate, fully-functional OStreams. Section 2.5.4.

Omodtest An interactive test program for running OModules. Section 2.3.8.

Oserver A program which allows O objects to locate classes on remote machines. Section 2.4.1.

SCHEME A programming language similar to LISP, with a very simple syntax. Section 2.3.1.

Stdio OModule An OModule class generated by an Oserver, which is handled by a program that takes input from stdin and sends output through stdout. Section 2.4.1.

X toolkit A library of objects used to form interfaces for X applications. Section 2.2.2.

Xiface module A special OModule class which allows quick generation of an X interface to running OModules. Section 2.3.8.

built-in OModule An OModule class which is defined by every O system. Section 2.3.4.

class registration The process of making a class known to an Oserver, thereby making that class available to all O objects. Section 2.4.1.

configuration information The information supplied to an OModule when it is created, which specifies the behavior of the OModule, according to the OModule's class. Section 2.3.5.

connection An O element which carries messages from the output of one OModule to the input of another OModule. Section 2.3.7.

delayinst module A special OModule class which can delay the instantiation of other modules. Section 2.5.3.

delivery mode The options selected for delivery of a message - currently includes immediate, queued, unregistered, and registered delivery. Section 2.3.6.

delivery mode filter A property of connections which allows a connection to affect the delivery mode of messages going through the connection. Section 2.3.7.

delivery notification output A special output on every OModule which emits a signal whenever a registered message from that OModule is delivered to its destination. Section 2.3.5.

hostname A name given to a machine on a network, which serves as part of a hostname/port address, used to address network servers. Section 2.5.1.

immediate delivery A message delivery mode which guarantees that a message will immediately be delivered to its destination. Section 2.3.6.

instantiation number An identifier for OModule instances, which is unique among OModules of the same class. Section 2.3.3.

message passing A method of communicating information and invoking actions between objects in an object-oriented system. Section 2.3.1.

modsend module A special OModule class which can send other modules over the network to be handled on a remote machine. Section 2.5.2.

module alias An OModule which represents another OModule. Section 2.3.4.

module group An OModule class which is defined by a group of OModules. Section 2.3.4.

object encapsulation The act of combining data and functions into a single unit, an object. Section 2.2.2.

port number A number used to address a specific network server on a specific machine. Section 2.5.1.

queued delivery A message delivery mode which directs a message to be sent to a queue, for later delivery. Section 2.3.6.

registered delivery A message delivery mode which causes the source OModule to emit a delivery notification message upon delivery of the message. Section 2.3.6.

resource A specific channel of communication used in the communication between toparent and tochild OModules. Section 2.6.2.

server/client A network communication arrangement in which one entity, a client, forms a single two-way connection with another entity, a server, and requests the server to perform tasks or deliver information. Section 2.2.1.

tochild module A special OModule class used in the Host/OStream model, which allows a Host to talk to an OStream. Section 2.6.2.

toparent module A special OModule class used in the Host/OStream model, which allows an OStream to talk to a Host. Section 2.6.2.

transportable objects The O premise that all objects can be moved to different machines, or split across different machines. Section 2.5.

uninstantiated module An OModule which has no class. Section 2.3.4.

unregistered delivery A message delivery mode which does not cause the source OModule to emit a delivery notification message. Section 2.3.6.

2.2 Story of O

2.2.1 Motivation

The basic premise of O is that distributed media in the computing environment no longer has to be static data which is transported from one place to another. Instead, the data can have instructions and methods included with it. What this means is that the behavior of the multimedia is determined neither by the sender or the receiver, but by the encoded behaviors embedded into the media itself.

This model arose to alleviate the time-consuming process of writing an entirely new encoder/decoder pair (see figure 2.1) for every new video coding scheme developed

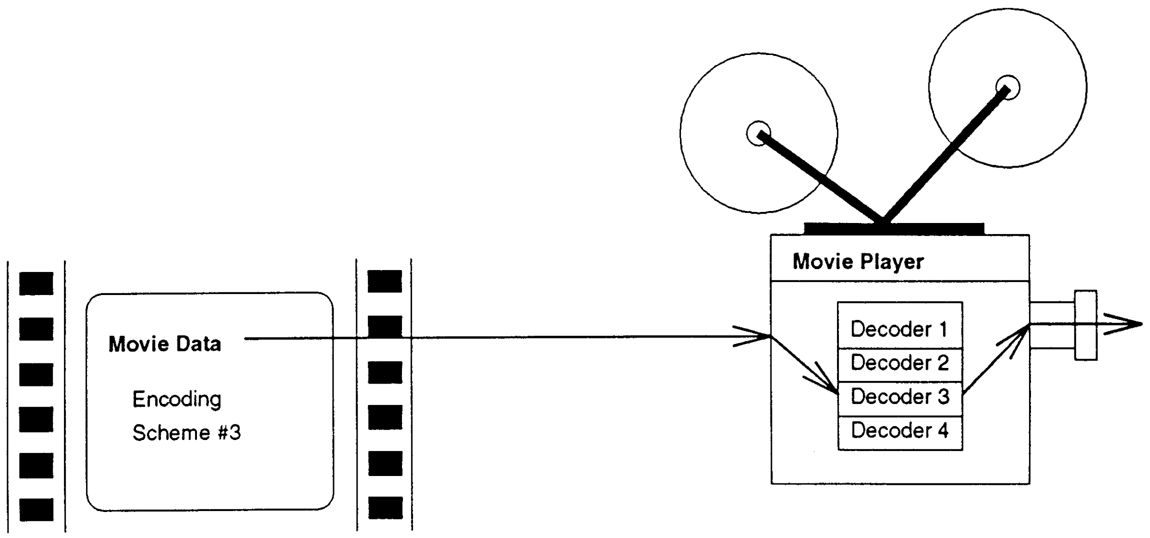

Figure 2.1: Movie player with many decoders

within our group (the Entertainment and Information Technology group at the Media Lab). What we really wanted was to write one universal server and client, and somehow embed the encoding and decoding instructions into the movie itself (see figure 2.2).

The idea of embedding instructions into a multimedia stream is not new. MIME [BF92], for example, specifies a multimedia representation for Internet mail. While MIME supports several encoding schemes, the specification also allows for an *application* to be included in the mail, where that application may serve as an interpreter for the rest of the mail. This is only one way in which behavior might be encoded in transmitted data. For our research, we have been conducting a structured exploration of embedded behavior through a series of stages:

Embedded decoding methods

This is the basic model already put forth, in which the encoded byte stream carries its own decoding instructions. These instructions are expressed in a universally established language. Every movie player will have an interpreter for this universal language.

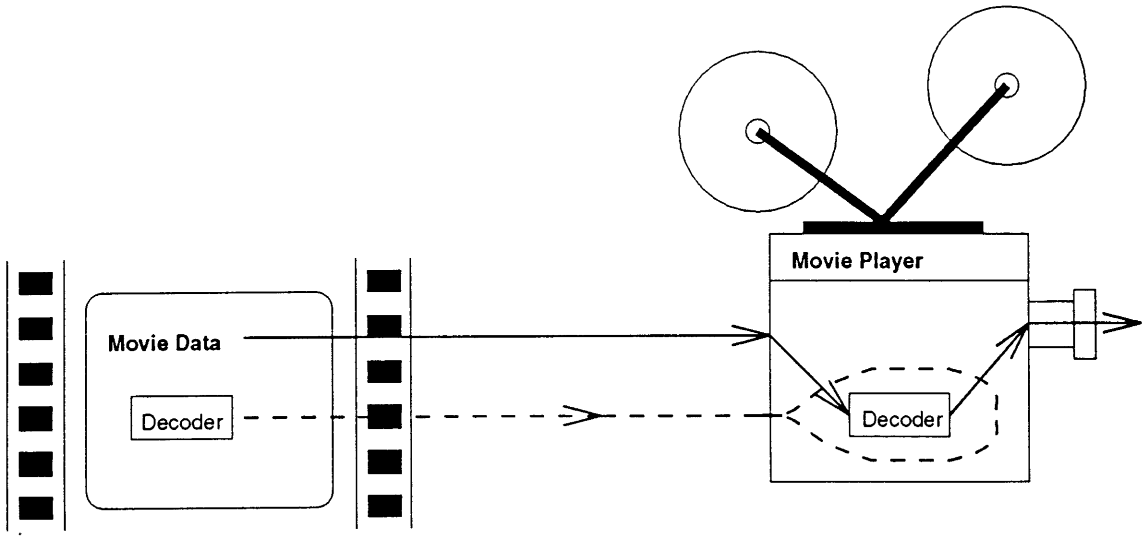

Figure 2.2: Universal movie player using embedded decoders

Embedded interaction methods

Traditionally the application program has had the burden of handling interaction with the user and interpreting that interaction into something which is used to directly manipulate the movie objects. With O, the movie's response to user interaction can also be embedded as a behavior that is interpreted by the universal movie player. This way, pressing the "rewind" button on a movie might do something different than pressing the "rewind" button on a multimedia newspaper, and the application doesn't have to know the difference.

This is where we enter the world of object-oriented multimedia. We can start thinking of multimedia as closed packages which have a few hooks that applications can adjust. Besides the explicitly stated hooks, applications shouldn't have to worry about what happens.

Transportable methods

With the addition of embedded interaction methods, we have moved our model of distributed multimedia from a byte stream to an object-oriented package. However the traditional object-oriented model says little about how an object might be split across two entities, such as a sender and a receiver. Thus the notion of embedded behaviors had to be developed carefully around the idea that some of these behaviors would have to be transported from one machine to another, that the object could actually exist on several machines at once.

In order to make this advance, we need an addition to the underlying architecture - the Oserver. On every machine that will be expected to "host" these multimedia objects, there must be some process listening and waiting for multimedia objects to arrive, knowing what to do with them when they do arrive, and of course there must be a protocol set up to support this model of an object with transportable methods.

Context-sensitive methods

The final stage of the O idea is a model built on the rest of O. In this stage, the series of adjustments that goes on between application and O multimedia object is formalized. The application explains to the O object what will be expected of it and what resources are available to it. The O object adjusts itself in turn, or perhaps negotiates with the application if it finds any of those terms disagreeable. O objects with this embedded behavior are called "context-sensitive", and applications which follow this model are called "contexts".

This final stage will require a protocol for negotiating an agreement between contexts and movie objects. This will also require something from the support system - a back channel of communication. Rather than having a single station broadcasting to many clients, each server/client connection will now have to be a two-way stream. While this may seem unreasonable in the present reality of broadcast television, it is actually not far off - already there are some cable companies experimenting with real-time feedback from their viewers. And in the future, media can be distributed on compact disks or over phone lines to private home servers (the VCR of the future).

2.2.2 Design goals of O

Once the basic ideas of O were put forth, a design was developed for the implementation of O, and also for how O would be visible to applications and users. The following design goals were observed:

Ease of use

One of the major problems with multimedia, and especially with networked multimedia, is the daunting complexity of working with any of it. Too many fields are encompassed all at once (networking, movie coding, etc.) that most people who start from the bottom never actually make it all the way to the top.

O is a serious attempt to take away all that complexity, to encapsulate everything in a model that is clear, easy to understand, yet still fully functional and configurable. This is O's primary goal, and many of the following goals are derived from it.

Object encapsulation

Object encapsulation (from which, incidentally, O derives its name) is one of the design goals of O. This means that object encapsulation will be used consistently through all levels of multimedia design, not just at the top application interface level.

Two (three) levels of programming

One of the keys to ease of use is to separate programming into two distinct levels - an application level and an object level. The application level is most straightforward and is designed for interacting with existing multimedia objects. The object level is more complex (and more functional) and is meant for creating multimedia objects.

This is similar to the X toolkit programming model [NO90]. Widgets are preconstructed objects, and the interface for using them is relatively simple. However, a lesson can be learned here, because writing widget objects is extremely difficult. Only a tiny fraction of X toolkit users are actually capable of creating a new widget.

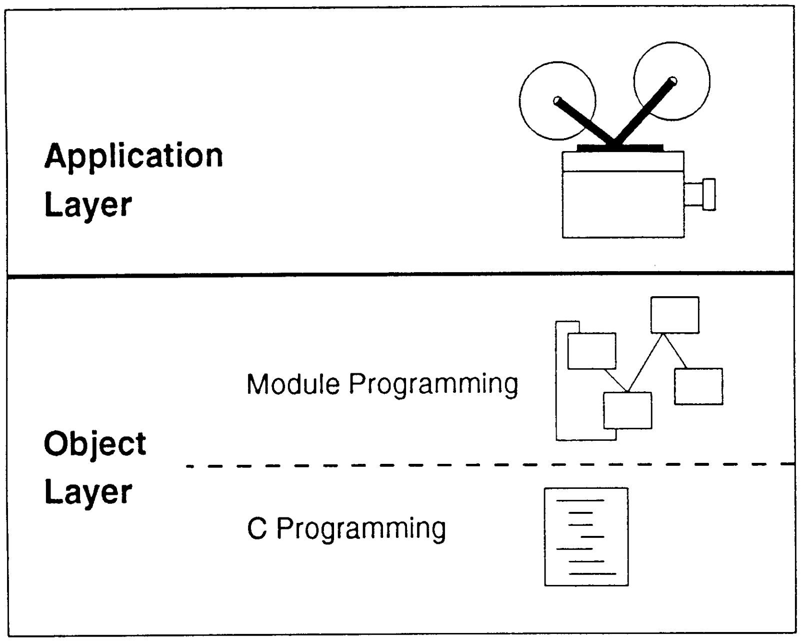

Figure 2.3: Levels of programming in O

This problem suggests an additional division of object programming into two more levels. The purpose of these two levels is to push all C programming down to the lower of the two levels, and make the higher of the two levels much easier to use. The trick is to balance ease of use with functionality, and also with efficiency in the real-time environment. Adhering to the object-oriented design goal, the higher of the two levels is achieved by providing a set of built-in objects that can be wired-together to produce the more complicated multimedia objects. The lower programming level can be used to create new low-level objects, but programming on this level should in general be unnecessary.

Network invisibility

The network is a daunting creature to most applications programmers, and it is one of the favorite things for big systems to abstract away (NFS, X, etc.). In the same tradition, O tries as hard as possible to make the network invisible, or at least model the network in a very simplistic way. This is true on all three programming levels mentioned above. For the application level, it is the most true, so that the application doesn't even have to know that there is a network out there. On the object level, the programmer must be aware that there are separate machines that the object may be running on, but that is modeled and abstracted so heavily that it can hardly be called network programming. Even on the C programming level, there are the Dtype [Abr92b] and dsys [Abr92a] libraries to keep networking as far away as possible.

This also means that all machine architecture differences have to be abstracted away, which is done mostly through the dsys and Dtype libraries.

Remote object location

One of the last things to be added onto O is remote object location. This feature enables O to locate objects which may be distributed at different sites throughout the network.

While this seems like a "fringe" feature for O, it is actually essential to achieving the network invisibility goal. This way objects can refer to other objects which may be local or remote, and O can properly locate these objects automatically.

2.2.3 Evolution of object encapsulation in O

As mentioned before, the object-oriented idea of embedding behavior into data is not new to multimedia. However, the idea of an object which can be split across several machines is tricky to develop into a design, as is the delicate balance between ease of use, functionality, and efficiency.

The primary question is, when someone wants to program an object, what description language will they use? How are the instructions embedded into the objects? The following explains what methods were tried as O evolved.

OSCRIPT

The first idea was to make the description language into a traditional language syntax, called OSCRIPT, based heavily on C. The reason was that such a language would be fully functional, it would be familiar to C programmers, and it could be run through a compiler that would turn it into something that could be interpreted efficiently.

Then, the transportable methods issue must be addressed. To do this, the language was broken down in such a way that entire functions could be sent across the network, so that a function called "sender()" might be running on the server machine, and a function called "receiver()" might be running on the client machine. Special constructs were included in the language for allowing the separated functions to communicate with each other.

However, serious problems arose with this approach. A C-like language, while fully functional, is problematic in that it is highly disorganized. This becomes very difficult to work with in a running distributed process. In such a situation, it is very important to keep track of who is causing what to happen when, how to send replies back, synchronization, etc. C, and in fact most traditional programming languages, are very bad at organizing real-time processes in this manner.

Omodules

The second method that was attempted was a graphically based process description language, similar to MAX [PZ] from IRCAM and Data Explorer [IBM91] from IBM. Here, specific functional elements (modules) are wired together (with connections) in such a way that it is graphically obvious how information is flowing. All activity occurs in a consistent manner, through message passing between objects.

With this model, the transportable methods issue becomes simple. Given a network of modules and connections, one can specify that certain modules should run on one machine, other modules should run on another, and the underlying system will automatically preserve the connections between the modules, even across the

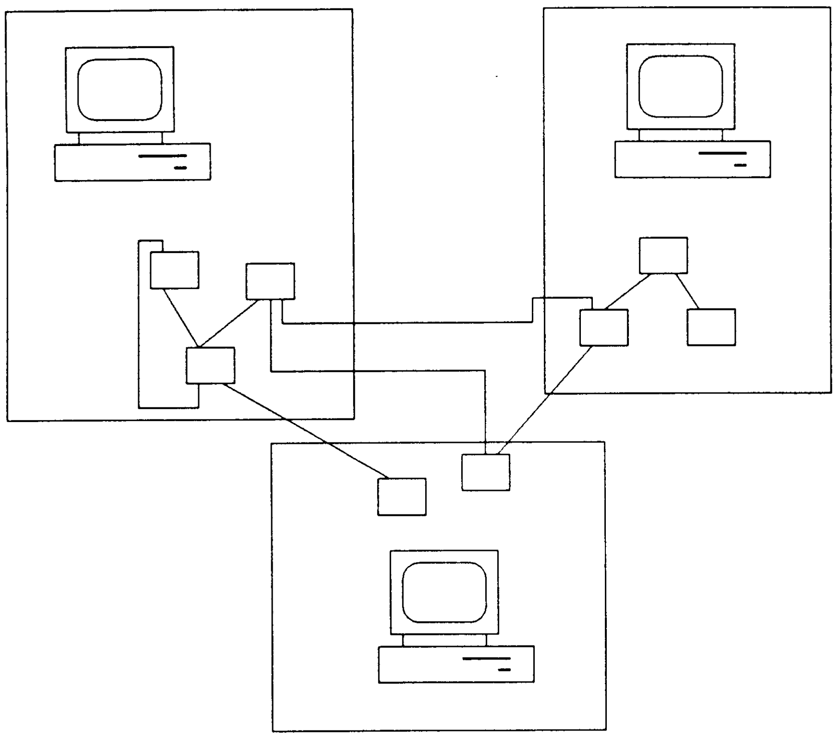

Figure 2.4: Module network split across machines

network, all invisibly to the programmer (see figure 2.4).

The problem with this approach is that while the high-level concepts are nicely packaged into powerful objects, the more mundane programming and housekeeping tasks become very tedious and difficult to write using boxes and lines.

Omodules and Evaluator

The first two approaches reveal the need for a third combination. The graphically based process could be used for establishing the high-level behaviors, while the programming details could be done in some more traditional programming language. This is the third and final approach. Here, the modules and connections concept remains the same as previously described. However, a new kind of module, an **evaluator** module, has been added. This module is fully programmable and configurable using a SCHEME-like syntax (which is similar to LISP). This language was chosen for many reasons (not the least of which is that it is trivial to interpret).

In the end, what we have is a clearly laid out network which describes the behavior of an object through the flow of information. The basic elements of the network are either preconstructed, or else modules whose behaviors have been described in a text-based SCHEME-like programming language.

2.3 Module networks

The basic model of describing the behavior of an O object is through modules and connections. Modules are computational elements with inputs and outputs. Each module has a specification which relates its outputs to its inputs. Communication into and out of a module (through the inputs and outputs) occurs through message passing. Messages are encapsulated in a standard method (Dtypes).

Connections are used to carry messages from the output of one module to the input of another module. A single output may connect to any number of destinations, and any number of sources may connect to a single input.

A network of modules is a set of modules and their connections. Figure 2.5 shows an example module network. This figure identifies all the elements of the network, many of which have not yet been discussed. These elements will be described later, and this figure can then be used as a reference.

2.3.1 Message Passing

Message passing is the basic means of communication between modules. Message passing is also the basic indicator and carrier of activity in the module network. Most modules do nothing until a message is passed to them, which causes them to process the message and send other messages as output. Connections carry the output messages to the inputs of other modules. Of course, there must be "instigator" modules to start the process. Such modules would send messages spontaneously, either according to a timer or else in response to some other activity on the system.

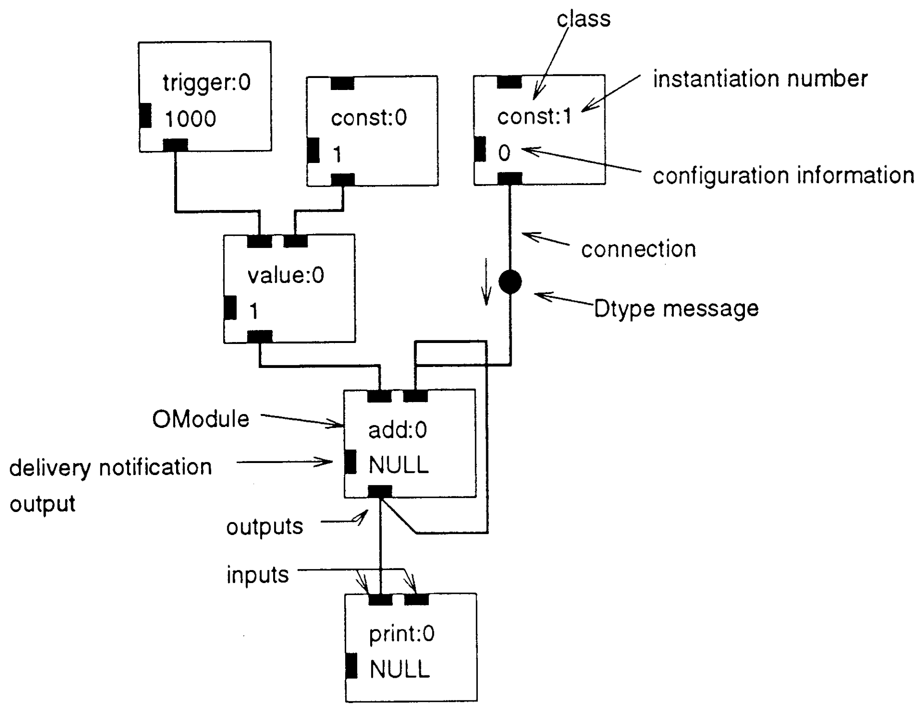

Figure 2.5: A module network

All messages are passed using a standardized data structure called a Dtype. The following is a brief description of Dtypes, but the Dtype manual [Abr92b] should be consulted for a full description of Dtypes.

Dtypes

Dtypes are typed data structures, which are used to represent many commonly used data types in a universal format. The data types supported include integers, reals, strings, variable-length lists, and packets of raw bytes. Dtypes are supported by a library of functions which construct, destroy, interpret, print, load, save, send and receive Dtypes. The following is an example of a Dtype:

```
(3 4 "hello" (2.4 -3.6))
```

This is a typical Dtype consisting of a list of integers, strings, and reals. The following code would generate this Dtype:

```
DtypeCreate(DTYPE_LIST,
            DtypeListCreate(DTYPE_INT, 3,
                            DTYPE_INT, 4,
                            DTYPE_STRING, "hello",
                            DTYPE_LIST,
                            DtypeListCreate(DTYPE_REAL, 2.4,
                                            DTYPE_REAL, -3.6,
                                            NULL),
                            NULL))
```

One of the biggest problems with distributed computing is that very few facilities exist for sending anything besides raw bytes between machines. Programmers of distributed processes often spend a lot of time writing routines that convert information into raw bytes that can be sent to another process, and routines that can convert raw bytes back into information. The Dtype library is an attempt to write this code once and set it as a standard that all programmers can use. The only real catch is that programmers must be prepared to express their information as Dtypes, and also be prepared to interpret Dtypes as information. But once programs are set up to use Dtypes, they can instantly send information back and forth through functions in the Dtype library.

The Dtype library also contains support for two important applications. The first application is Dtype servers, which are servers that communicate with their clients through Dtypes. The Dtype library has enough support for this application that setting up a fully-functional multiclient Dtype server takes a couple of hours at the most. The second application is the Dtype evaluator, which can interpret Dtype lists as expressions or programs, in the style of LISP or SCHEME. The Dtype evaluator is a lightweight interpreter for such expressions, which is useful in a situation where an interpreted language needs to be "thrown in" to an application.

2.3.2 OModules

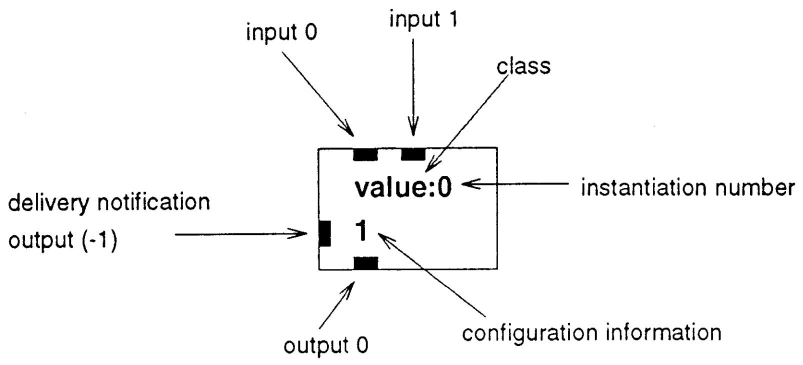

Figure 2.6: An OModule

An OModule is the basic unit of the O system. An OModule can be visualized as a box, with a number of inputs and outputs (see figure 2.6). What is inside the box can be any number of things, but no matter what is inside the OModule, or for that matter no matter where the OModule is located on the network, the interface to the OModule and the behavior of the OModule remains consistent.

As a point of reference, the inputs and outputs are numbered in order, from left to right, starting with 0.

2.3.3 OModule classes and instantiations

An OModule's class is what determines the behavior of the OModule. All OModules in the same module network which have the same class name will all have the same behavior. The behavior is determined by the handler which is assigned to the OModule. The process of translating OModule class names into OModule handlers is called "module instantiation". The process of module instantiation is a little tricky, since the handler assigned to an OModule class depends partially on the environment of the OModule - i.e., which module network it is in, which machine it is on. In general, the module instantiation process is pretty clear, with a few oddities.

Every OModule also has an instantiation number, which is used to differentiate two OModules of the same class. Within a single module network, no two OModules of the same class should have the same instantiation number. Therefore, within a single module network, a <class>:<instantiation> name is enough to identify a specific module.

2.3.4 OModule types

As mentioned earlier, what behavior, or handler, is put into an OModule box is determined when the OModule is instantiated. After that, it doesn't matter on the outside of the OModule what is actually inside the box.

However, to get a handle on what's going on, it is often useful to know what is inside the box. There are three basic OModule types, one of which is selected when the OModule is instantiated.

Built-in modules

The built-in module classes are the module classes that come prepackaged with every O system. Any module which bears the name of one of these built-in classes will automatically have the handler from that class assigned to it. These handlers are actual C subroutines that are included in the O library, and a catalog exists which documents the behavior of each of these classes. In most cases, these modules are very simple, such as modules which add, or hold values, or send out timing pulses at regular intervals. However, these modules are basic and powerful enough to do just about anything when wired together in the right combination.

Module groups

Module groups are the first level of abstraction in O. Once a network of modules and connections has been made, that entire network can be turned into a module group if it is given a name. If an OModule uses that name as its class, then its handler will be that entire module network. Of course, we must emphasize again that to whomever is using that OModule, the fact that there is an entire module network inside that OModule is completely hidden. It looks just like any other module.

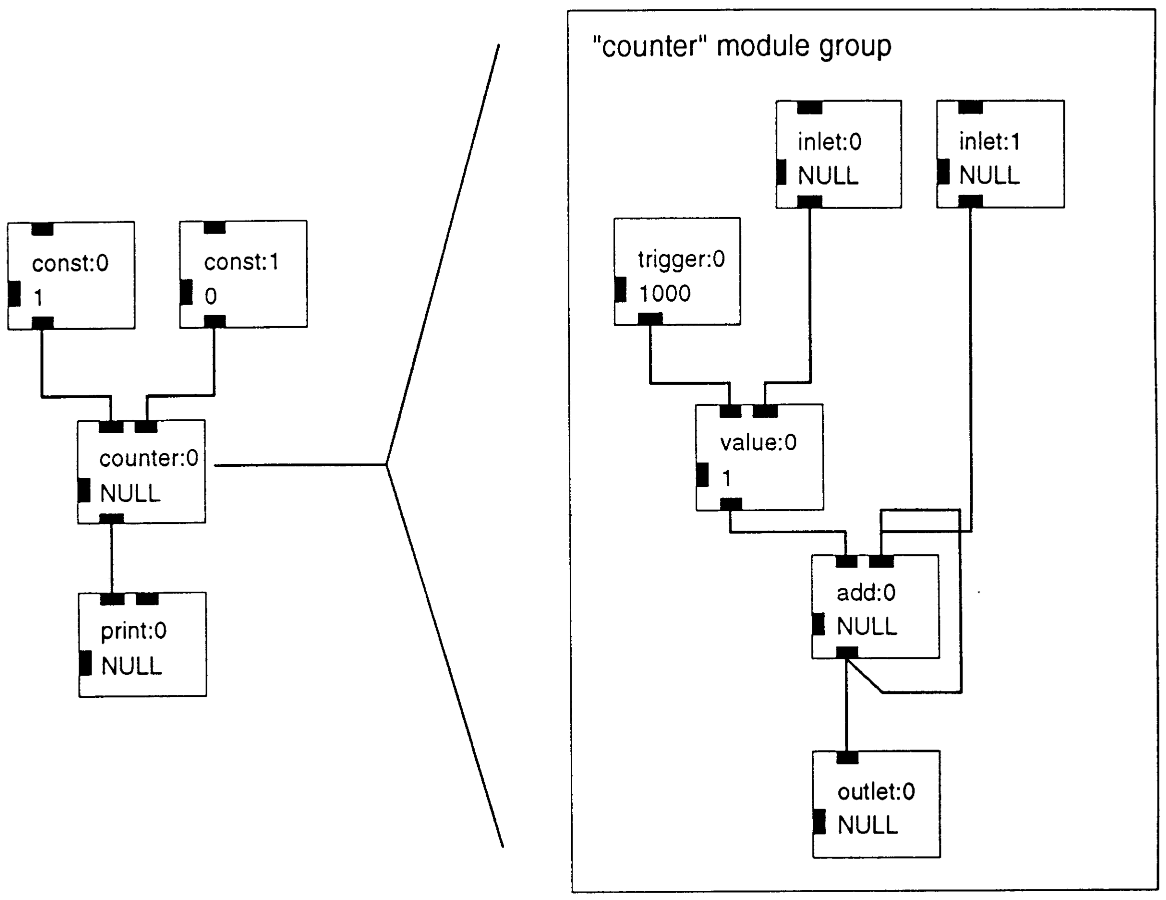

Figure 2.7: A module group

Just like any other OModule, a module group receives messages through its inputs and sends messages out through its outputs. To the module network which defines the module group, the inputs and outputs are represented by special built-in modules called inlets and outlets. Messages which go in to the OModule are passed into the module group through inlet modules. Messages which the module group sends to outlet modules are passed through the output of the OModule.

Figure 2.7 shows a module group being used in a module network. Note the inlet and outlet modules in the module group, and their relation to the inputs and outputs of the "counter" module.

Some terminology - when a set of modules and connections is represented by a

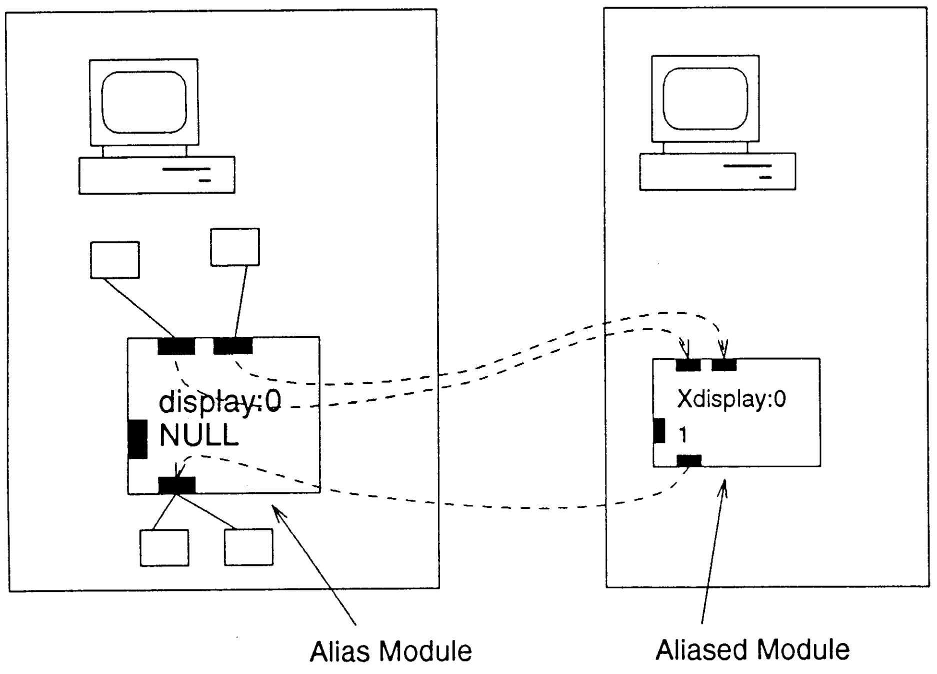

Figure 2.8: A remote alias module

single OModule, that OModule is called the "parent" of the underlying modules, and those underlying modules are the "children" of the OModule. There is also no limit to the nesting of module groups - a module group may contain modules which are themselves module groups, etc. Also, note that from now on "module group" and "module network" will be used interchangeably, since they both refer to the same concept[1].

Module aliases

Module aliases are the means by which one OModule can represent another OModule. An OModule alias always has a target OModule which is what the original OModule aliases to. Any messages sent to the aliased OModule are rerouted to the target, and any messages originating from the target are rerouted back through the outputs of the original OModule (see figure 2.8).

[1] "Module group" usually implies a module network that is being used as a single module within another module network

OModule aliases are the mechanism for hiding the fact that an OModule may exist on another machine. In that case, whoever is using the OModule can still send it messages just as if it were a normal OModule, but because the module is actually an alias to some remote machine, messages are automatically and invisibly rerouted to and from the target OModule.

Uninstantiated modules

This is actually more of an OModule non-type. An uninstantiated module is completely nonfunctional - messages sent to it do absolutely nothing, and it generates no messages.

All modules are uninstantiated when first created. Sometimes it is desirable to leave a module uninstantiated. This occurs most often when there is a module that is going to be sent to a remote destination - in this case, there is no point in instantiating the module until it gets to its destination, so the module remains uninstantiated until it is sent.

2.3.5 Configuration information

Every OModule, in addition to being identified with a class and an instantiation number, also has configuration information. This is a single Dtype which the module uses to control its operation. Every module class will use the configuration in a different way. For example, a module which "ticks" at regular intervals might expect the configuration information to be an integer specifying the tick interval. A general convention is to have the configuration information be an initial configuring value, but also to use one of the inputs to allow that value to be changed during runtime.

For the built-in module classes, what configuration information is expected for each class and how that configuration information is used is included in the built-in module catalog listing. When the module network is created by a designer, the configuration information for each module in the network is also specified at that time by the designer.

Inheriting configuration information

According to the description above, the configuration information for each module is a static field which is determined at the time the module network is designed. However, it may be desirable to allow that configuration information to change at runtime. Here is an example:

Suppose a module group is created to output numbers at regular intervals. Such a module group would require a few modules and a ticker module at the center. The configuration information of the ticker module determines the rate at which the numbers will be sent out.

Once this module group has been created, it can be treated as a single module and incorporated into other module networks. However, these module networks have no control over the timing interval of the module group, since the timing interval is preset as the configuration information of the ticker module, which is hidden away in the module group.

What is needed is a way to pass configuration information into a module group, and have that configuration information be distributed to the proper modules. In a sense, the module group itself needs to be configured. This fits perfectly, since a module group appears to be a single module, and like any other module it gets its own configuration information. So, the issue is how to pass configuration information specified for the module group down to the individual modules in the group - how is configuration information inherited through different levels of the module group hierarchy?

The O answer to this problem is to use a feature of the Dtype library called **environments** and **Dtype references**. A Dtype environment is simply a list of keyword/value pairs, such as the following:

```
((ticktime 2000) (color red) (WMBR 88.1))
```

The configuration information for a module group is such an environment.

The complement of a Dtype environment is a **Dtype reference**. A Dtype reference is a way of "looking up" values in a Dtype environment. A Dtype reference has the following form:

```
(@ ticktime)
```

When processed through the above environment, the entire Dtype reference is translated into "2000". The Dtype reference process is also recursive, so that if the following Dtype were passed through the environment:

```
(20 30 (@ ticktime) 3.5)
```

it would be translated into the following:

```
(20 30 2000 3.5)
```

The Dtype environments and Dtype references are used to allow modules to inherit configuration information from their parents. When a module is instantiated, its configuration information is first passed through as a Dtype reference, using the parent's configuration information as a Dtype environment. This way, any configuration information parameters that should be inherited are specified as Dtype references, and when the module group is included in a module network, those parameters should be specified in a Dtype environment as the configuration information to the module group.

For example, refer to figure 2.9. This shows the same "counter" module group example as before, except now the module group's "ticker" gets its timing parameter from the parent's configuration information.

This process is extended to work even when module groups are nested several layers deep. When a module uses a Dtype reference to specify a parameter, the search for that parameter first occurs in the configuration information of the parent. If not found there, the search continues to the grandparent, then the level above that, etc. If not found in any configuration information, then the parameter is set to NULL.

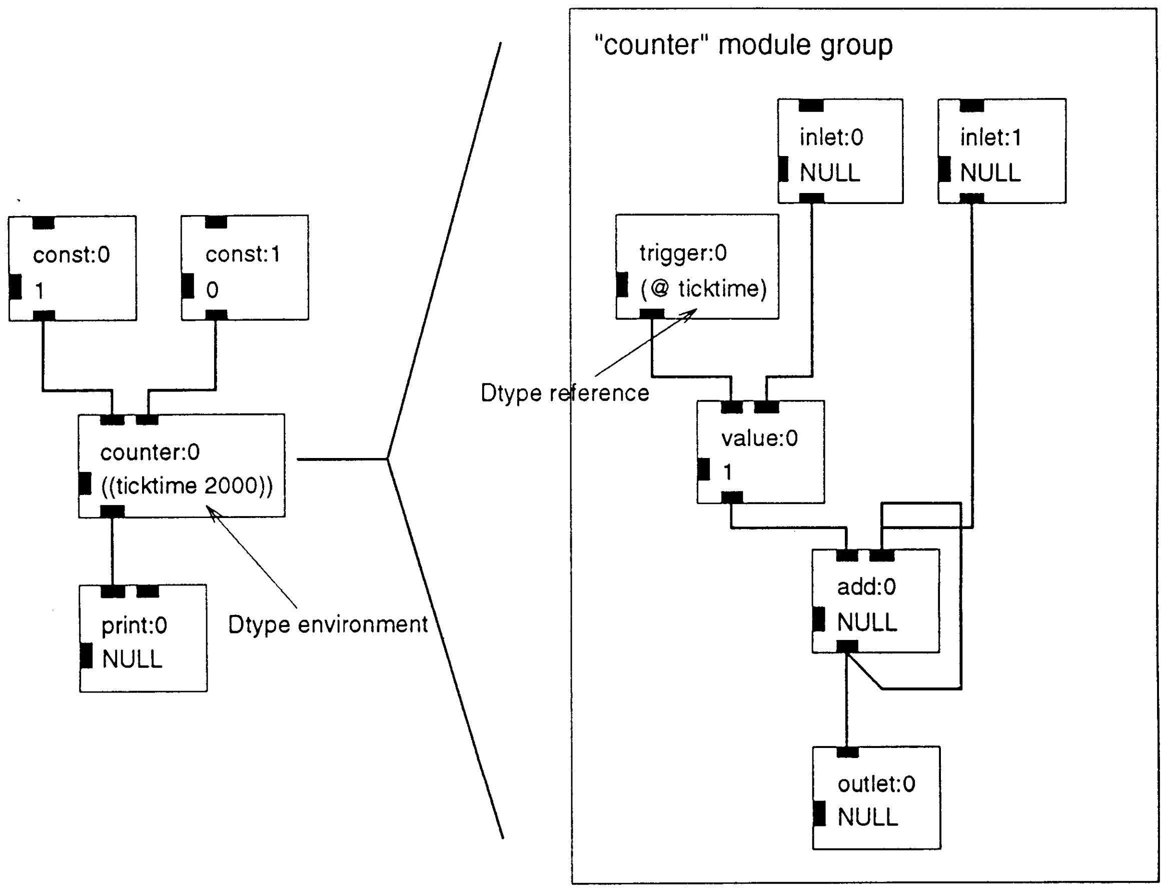

Figure 2.9: A module group inheriting configuration information

The delivery notification output

Sending messages from an OModule is not necessarily a synchronous process. Messages may be delayed in a queue, they might be delayed traveling through the network, etc. It is often difficult to tell exactly when a message has arrived at its destination.

Sometimes it is useful to know when a message has been delivered to its destination. Take for example an application which is trying to send video frames from a source to a destination as fast as possible. The source will want to know when the frame it just sent to the destination arrives, so that it can immediately send out another one.

To handle this situation, every module includes an extra output called the *delivery notification output.* Visually this is represented as an output on the left side of the module - out of the way since it is rarely used, but always available.

The purpose of a module's delivery notification output is to signal every time a *registered* message sent by that module has been delivered. Whenever a message sent by that module is delivered to its destination, an integer is sent out the delivery notification output. The value of that integer is the output number that sent the message.

Not all messages are registered. By default, messages are unregistered, but that status can be modified by the connections, as explained later.

The delivery notification output, like all other outputs, has a number, which is -1.

The Eval OModule

The basic set of built-in OModule classes is sufficient to perform just about any computation. However, in actual use it was awkward to build a useful module network out of such small pieces, somewhat akin to building a computer purely from logic gates.

The electronic hardware solution to this problem is a logic device called a PAL, for Programmable Array Logic. On the outside, a PAL is just a normal logic device.

On the inside, however, is a tiny bit of software which configures how the inputs and outputs of the PAL behave. This software can be easily changed to fit different applications, which makes the PAL act like a universal logic device.

The O analog to the PAL is called the Eval OModule. On the outside, the eval module is just like any other built-in module class. However, within the configuration information of the eval module is a program which describes how the inputs and outputs of the module behave. This program is written in a SCHEME-like language which is handled by the Dtype evaluator (described in the Dtype manual). The reason for choosing this syntax is that it is simple to learn, simple to interpret, and trivial to fit into the Dtype syntax which is required for configuration information.

As an example, consider a module takes in 4 numbers, adds the first three and multiplies the result by the fourth and sends the final result out output number 0. Without using an eval module, this module would best be written as a module group with 4 inlet modules, 1 outlet module, 3 add modules and 1 mult module.

Or, this could be done in a single eval module with the following configuration information:

```
(1 0 ()
  (out 0
    ("*" ("+"
   (in 0)
           (in 1)
           (in 2))
      (in 3))))
```

In case this syntax appears foreign, here is a quick explanation - for every expression contained in parentheses, the first element of the list is a function name, the remaining elements are arguments. The entire list is evaluated by evaluating the arguments, passing them to the function, and what the function returns is the result of the evaluation. Some functions have side-effects, such as the "out" function which causes its second argument to be sent out the output number specified by the first argument.

The practice, an eval module usually serves as the central control unit for a module group, where status messages go into the module and control messages go out. This way, the behavior of the module group can be coded in something closer to a programming language, but the data flow in the module group is still represented through modules and connections.

2.3.6 Message passing and delivery modes

When a module sends a message out one of its outputs, several things happen before that message is actually delivered to its destination. First, the message's *delivery mode* is examined.

The delivery mode of a message is a set of options which control how the message is routed. One of these options is whether the message is to be delivered immediately or whether it is to be placed in a queue for delivery later. These delivery modes are called *immediate* and *queued* (see figure 2.10).

A message marked for immediate delivery will arrive at its destination as soon as it is sent. The destination module is then given a chance to process the message, perhaps sending out messages of its own. Only when the destination has fully finished processing the message will the original module regain control. This is important in two situations: first, if a message needs to be sent to two or more destinations, then the second destination will not receive the message until the first destination has completely finished its processing and generating its own messages. Second, if a circular loop has been formed in the module network such that a module is somehow connected to itself, then a message sent by that module might cause an endless loop and crash the O system. Despite these peculiarities, the immediate mode is attractive because of its simplicity and the predictability of its operation.

Conversely, a message marked for queued delivery will not be sent immediately to its destination, but instead the message is put onto the back of a message queue.

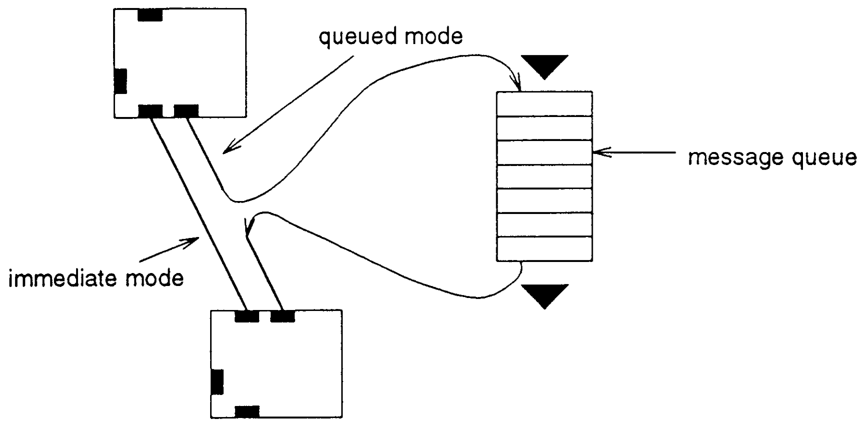

Figure 2.10: Immediate versus queued message delivery

When the O system gets a free moment (i.e., when it is done handling messages marked for immediate delivery), it takes a message off the queue and delivers it to its destination, where it is immediately processed. Messages are taken off the queue in the order that they are put on. The main applications for the queued delivery mode are to force a message to be delivered after a set of messages marked for immediate delivery, or to avoid crashing the O system in the event of a circular loop in a module network.

The queued/immediate choice is one delivery mode option, which defaults to the immediate delivery mode. The major exception to this default is the eval module, which sends out all its messages marked for queued mode. The reason for this is that since the module is often used as a central control point, the outputs of the module are sometimes routed back into its inputs. The queued default mode keeps this from becoming a problem without the user having to think about it.

Another delivery mode choice, which has been mentioned before, is the *registered/unregistered* option. This option affects the delivery notification output of the module which sent the message. If a message is sent unregistered, then nothing happens with the delivery notification output. If, however, the message is sent registered, then once the message is finally delivered to its destination, and after the destination has finished processing it, a message is sent out the sender's delivery notification output, as described previously. The default is for messages to be sent unregistered.

2.3.7 Connections

Connections are the objects which carry messages between modules. A connection is attached at one end to exactly one output, and at the other end to exactly one input. The output and input are allowed to be on the same module. More than one connection may be attached to a single input or output. An output may also have no connections attached to it. Messages may still be sent out that output, but they will not go anywhere.

In general, connections will only exist between modules which are in the same module network. Theoretically it is possible to connect any two modules anywhere, but use of that "feature" quickly leads to confusion, and no provision is made for it in the application interface.

Connections are equipped with delivery mode filters, which will affect the delivery mode of messages passing through the connections. By default, the filters are off, so that the delivery modes remain unaffected. When the connection is defined, it may be defined with its queued mode filter on, which means that any messages passing through are marked for queued delivery, no matter if they were marked for queued or immediate when the module sent them out. The same is true for the immediate mode filter, the registered mode filter, and the unregistered mode filter.

2.3.8 Module networks in operation

Once a network of modules and connections has been defined (probably through a graphical interface), a special interpreter is needed to breathe life into the network and start it running.

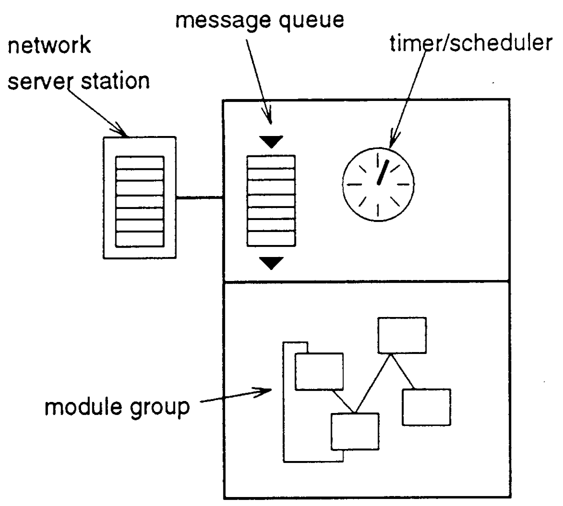

Figure 2.11: An OStream

OStreams

OStreams are the objects in O which are responsible for running module networks. Usually a single OStream will run a single module group (which may of course contain its own module groups, all still handled by the same OStream). The OStream is responsible for carrying messages from a module to their destinations or to the message queue. The OStream also maintains the message queue and is responsible for making sure that messages get off the queue and to their destinations as quickly as possible. The OStream takes care of "waking up" modules that require periodic processing, such as the "ticker" module. Finally, as will be described later, the OStream is a complete network server. This allows OStreams all over the network to talk to each other, for modules in different module groups on different machines to send messages to each other, and even allows entire modules to be sent from one machine to another. Figure 2.11 shows a figurative representation of an OStream.

Each OStream is implemented as a separate UNIX process.

Omodtest

The basic program for running and testing module groups is **Omodtest**. This program starts up an OStream, loads a module group (specified on the command line) into it, and starts the OStream running, thus running the module group. At the same time, the program accepts keyboard input, which allows the user to "poke in" to the module network and send messages to any module at any time. This is extremely useful for testing, and for finer inspection there is a debugging mode in which every message that is sent between two modules is printed to the console, along with the source and destination of the message.

Interfacing with Xiface

The Omodtest is a powerful but clumsy method for real-time interaction with a module group. It serves as a good debugging tool, but not as a nice user interface for a running O application.

O does have a provision for quickly creating a graphical user interface to a running OModule network. This is done, of course, through a special O built-in module class called an Xiface module. Currently, this Xiface module has been interfaced only to X, but it is theoretically possible to port the module to whatever platform it happens to be running on.

A separate X window will be created for each Xiface module that is instantiated in a running module group. The X window can contain buttons, text fields, menus, sliders, free-form line drawings, and even color images. What exactly is in the X window is determined by the configuration information passed to the Xiface module.

Interaction with the X window is mirrored through the inputs and outputs of the Xiface module. For example, if a button is pressed in the X window, then a message is sent out one of the outputs of the Xiface module. On the other hand, inputs to the module could be used to change text fields, display new images, etc. The specifications of all these arrangements are very straightforward and fully documented in OXiface.c.

Keep in mind that the Xiface module is an excellent way to quickly construct an X interface, but that interface is not as aesthetically and functionally configurable as a full application interface should be. For a full-fledged application which uses O, a separate interface scheme should be used.

2.4 OSpace

Running module groups and connections is only half of the Story of O. One of the stated purposes of O is to act as a distributed object system, while at the same time keeping the network invisible to the user. This goal fits in easily to the model of modules and connections. Two modules might be on the same machine or on different machines, and connections between them are preserved invisibly whether the modules are local or separated by a network.

This is the concept of *OSpace*. All modules on all the machines in a given network are considered to exist within a single universal set, called the OSpace. Any two modules within the OSpace can be connected together, and O will take care of routing messages through that connection from the source to the destination, even if messages have to go through the network. This way, the network remains invisible to anyone who is using O.

OSpace also includes the idea of distributed object class location. Not every machine has to know about every module class that it may be required to run. Every machine running O knows about the built-in module classes, but beyond that, new module classes may be created that some machines know about and others do not. This is where the distributed object class system comes into use. If a machine is asked to run a module group which contains a module whose class it does not recognize, it broadcasts a call for help to all other machines on the network. If some other machine knows how to handle that class, then an instance of that class is started up on the more enlightened machine, and the original machine reroutes connections to that module so that they go to the new machine. In other words, the module on the original machine becomes an *alias* for the module on the new machine.

There is an important point to be made here: when this situation occurs, the module group is now being distributed across two separate machines. There are other options for handling this situation which were considered but rejected. One option is to turn control of the whole module group entirely over to the enlightened machine, but this would not work if the group also included a module that the enlightened machine did not recognize, which would cause the module group to "ping-pong" back and forth between two machines. A second option is for the enlightened machine to somehow send a definition of the module class to the deprived machine, so that the deprived machine would then know about the class forever and not have to distribute its processing. The problem with this is that sometimes the distributed processing is desired, such as for efficiency in a parallel computation. Another problem is that sometimes there is a specific reason why a class might exist only on certain machines - those machines might have resources that other machines would not have, so it would not make sense for the class to be run on machines that are not properly equipped.

Ideally, the scheme for handling distributed object classes would be based on some intelligent combination of the above proposed solutions, since one option may be more efficient that the others in certain situations. However, for simplicity, the current design of O always distributes its processing of remote object classes.

Note that when objects, instances, and classes are mentioned, they are still referring to modules and module classes. By constraining the distributed object system to fit into the model of modules and connections, the network invisibility issue is resolved since the user has to make no special provisions for knowing whether a module will run locally or remotely.

2.4.1 Oservers

The *Oserver* is the basic element of the distributed object location system. Every machine that has publicly available object classes must run a single Oserver. The object classes that are available on the machine are *registered* with the Oserver. The registering process consists of stating the class's name, and specifying a *handler* for the class. When another machine broadcasts a request for a class, the Oserver checks its list of registered classes and responds if that class has been registered. The original asking machine then sorts through all the responses it received and chooses one Oserver to actually handle the class. Currently the choice is based on which Oserver is running on the least loaded machine.

The Oserver also handles creating an instance of a class for use by a remote machine. Once a machine has selected an Oserver to handle a class, the Oserver is directed to create an instance of the class, connect it to the original machine, and run the instance. The Oserver does this by forking off OStreams, which as explained earlier are capable of running modules and connecting modules across the network. At this point, the job of the Oserver is finished. Its responsibilities begin when someone is looking for a class which it knows about, and its responsibilities end once an instance of that class has been started. The OStream can handle things from there. Figures 2.12 and 2.13 show the process of a remote class being located and started using Oservers.

As mentioned before, all the objects started by an Oserver must follow the module model in form, and once they are running they all behave and interact like modules. However, the handlers for those objects can take on many different forms. Currently, the Oserver recognizes four different kinds of handlers: groups, stdio, executable, and Dtype client.

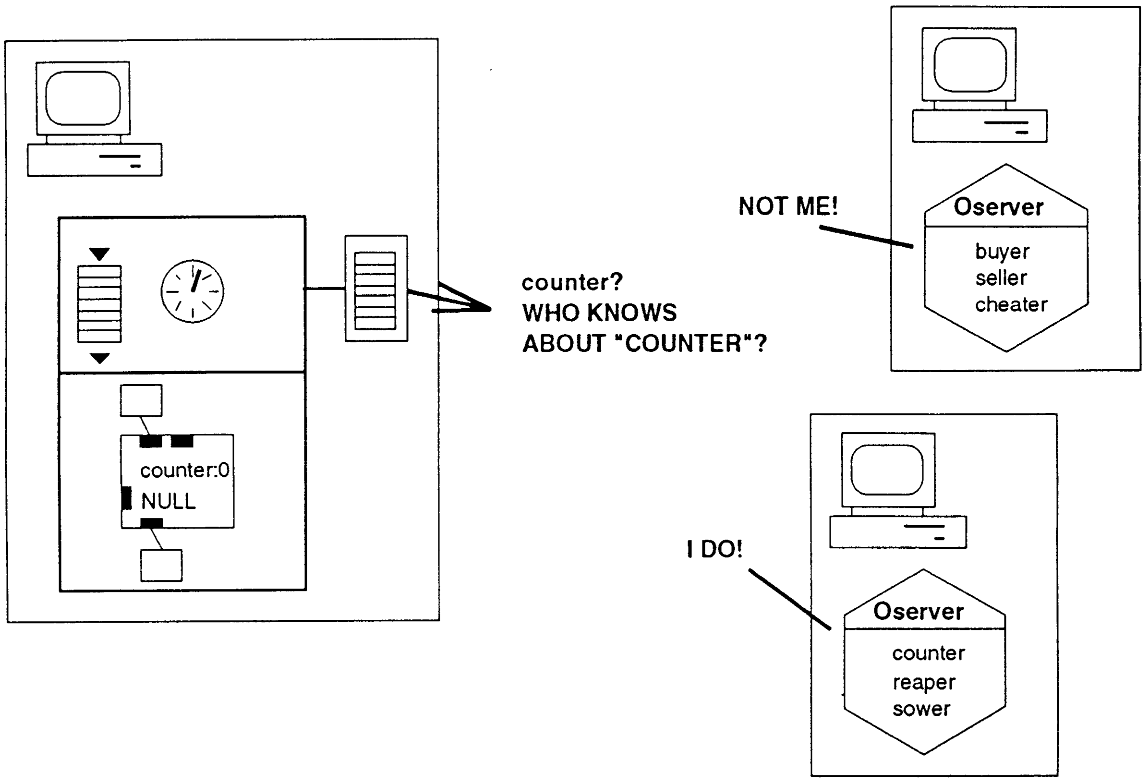

Figure 2.12: Remote object location using Oservers

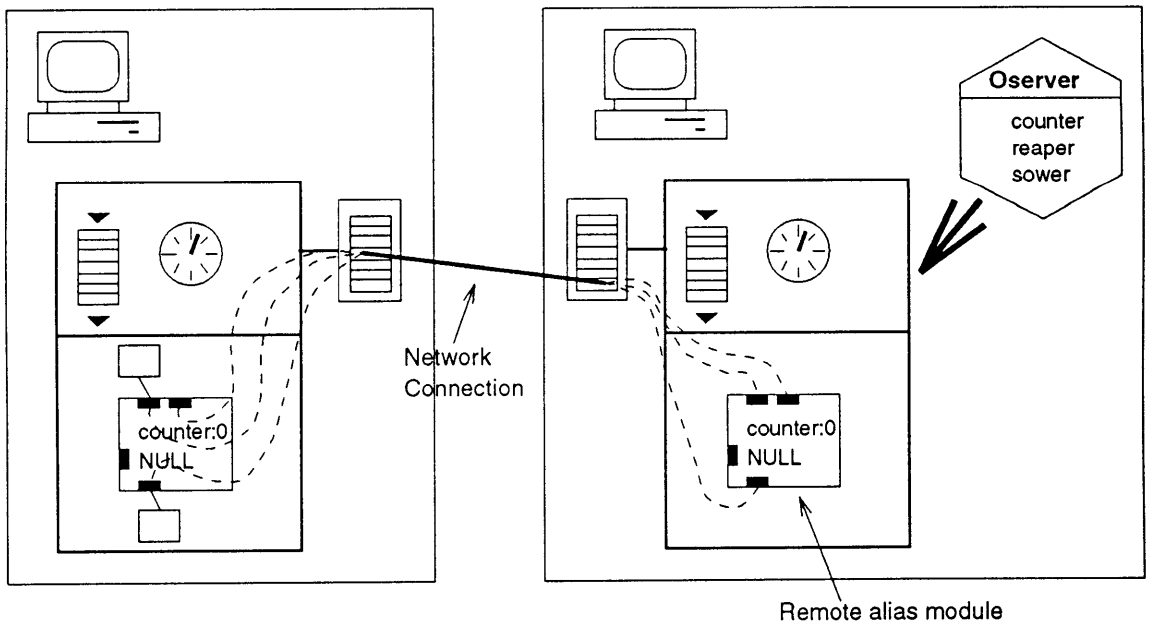

Figure 2.13: Oserver starting an OStream

OModule Groups

When a module network/group is created, it is saved to disk in a special formatted file, which may be registered with the Oserver as a handler. When the Oserver runs this module group, it will behave exactly as if it were being run by the Omodtest program, except that its inputs and outputs will be connected to a remote machine.

The module group handler is the easiest way of distributing an existing O object, since module groups can be separated from the object and placed on different machines.

Stdio OModules

Sometimes a special module class is needed quickly, a class whose behavior is most easily written in C and not easily described in a module group. A stdio module handler is provided for this situation. Such a handler is simply a C program which takes Dtypes (in ASCII form) from stdin, and sends Dtypes out (in ASCII) to stdout. The Oserver treats such a handler as a module with a single input and a single output. Messages which are sent to the input are converted into ASCII form and routed to stdin of the handler. Any output from stdout of the handler is captured and parsed into a message which appears at the output of the module (see figure 2.14).

O actually goes through a lot of work and spends a lot of time on the rerouting and parsing for a stdio module. However, the payoff is that the module handler is trivial to write, since the ASCII Dtypes are extremely simple to parse (especially if the module expects only numbers or strings). This is an excellent starting point for users who wish to write module handlers for O.

Executable OModules

Executable module handlers are the next step up in complexity from the stdio module handler. Like the stdio module handler, an executable module handler is a standalone program which the Oserver invokes and communicates with. However, the executable

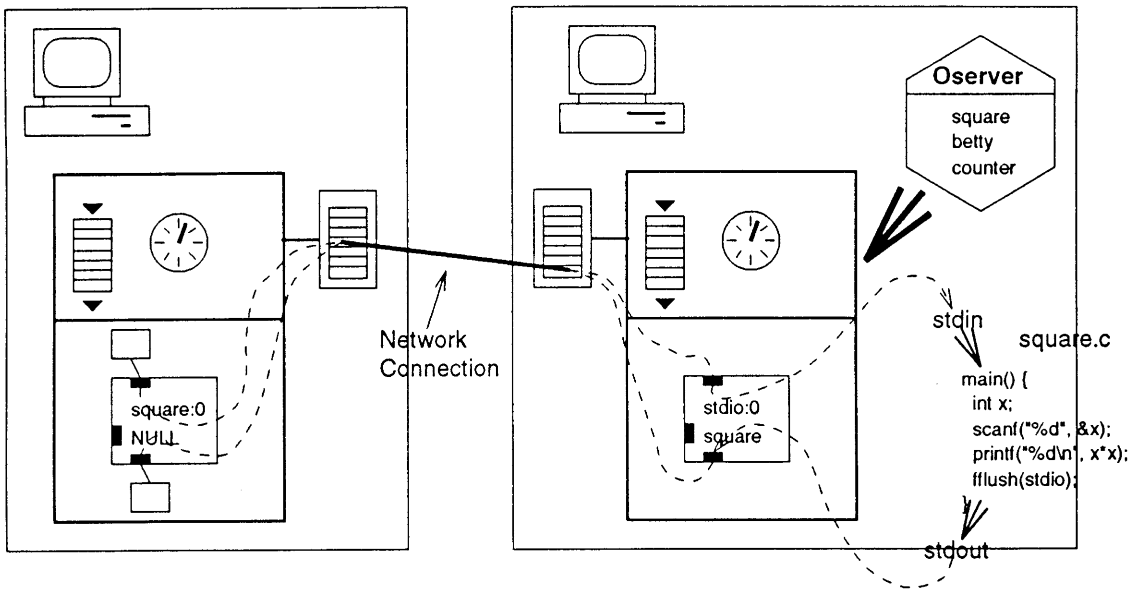

Figure 2.14: Stdio OModule class type

module handler has no limit on its number of inputs and outputs, and O is freed from a lot of the parsing and rerouting that is required by the stdio module handler. In return, the executable module handler is a little harder to write and must follow a special form. The procedure for writing an executable module handler is similar to writing a built-in module handler, and is described in the O User's Manual.

Dtype Client OModules

The Dtype package includes a method for quickly writing network servers, and recently several such servers have appeared in the Entertainment and Information Technology group - **betty** [Blo92], a news server, **videovisual**, a visual activity server similar to the UNIX "finger" command, **digitizesrv**, a server for digitizing video from a NeXTDimension board, **vpr3srv**, a server for remotely controlling a VPR3 1" video machine, and **doppelsrv**, a forthcoming user-modeling server. All of these servers are related in that the protocol for communicating with them is highly specialized. The servers receive messages in the form of Dtypes, and send Dtypes back

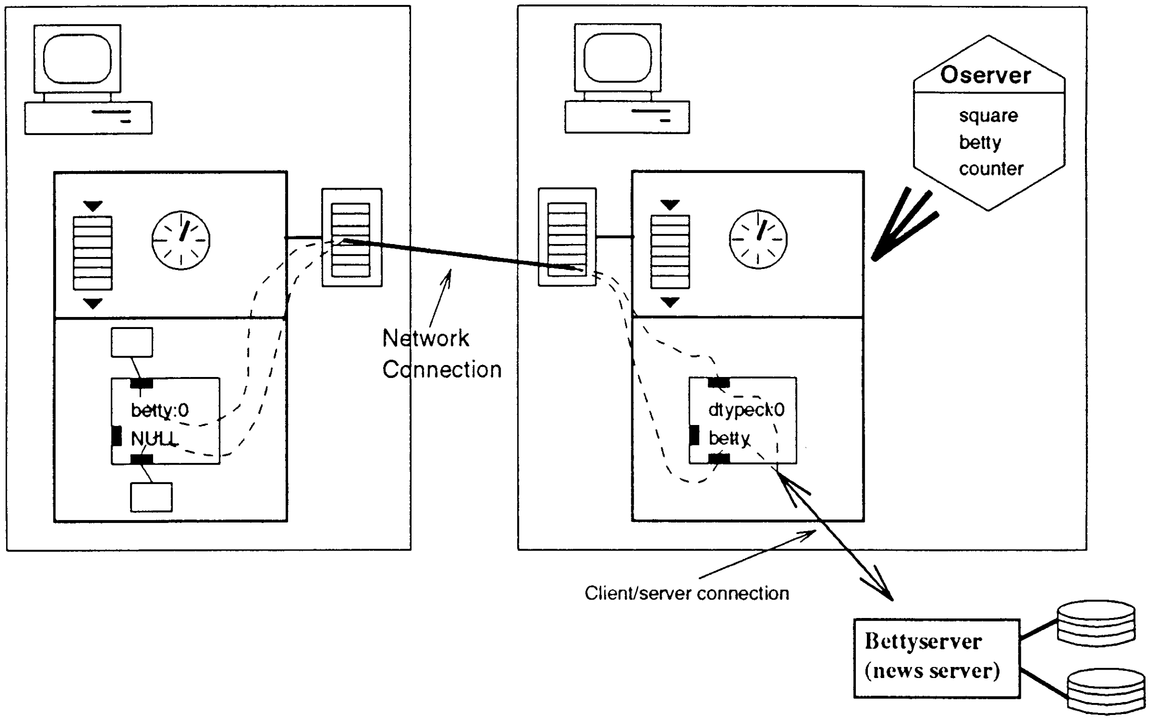

Figure 2.15: Dtype client OModule class type

as replies.

O includes a method for interfacing with any Dtype server. To O, a Dtype server can be modeled as a module with one input and one output (although a Dtype server can handle multiple clients at once, one client usually maintains only a single two-way connection with its server). The Dtype client module handler simulates a client connection with a Dtype server, by passing messages it receives to the Dtype server, and sending the server's replies back through the output of the Dtype client module (see figure 2.15).

To register a Dtype client module handler with the Oserver, the Oserver only needs to know the hostname and port (or service name) of the Dtype server. Any machine then asking for that class will be connected and routed to the Dtype server. With this mechanism, all the Dtype servers that are written or will be written are immediately available for all O objects to use.

2.5 Transportable Objects

So far, O has been presented as a means of running a process built from modules and connections, and O has also been shown as a network-based system with the ability to distribute different object classes among several machines. So far, many of the choices in the design of O may seem strange or arbitrary. However, all of these choices have been building up to support the final stage of O, its main purpose, the ability to transport objects.

Remember that O began as a way of encoding movies so that any movie player could understand and decode any encoded movie. Early on, it was decided that the movie would have to somehow carry its decoding methods along with it as the movie made the trip from the server to the movie player. This left two options open. The first option was to extrapolate from the traditional broadcast model and send the movie over as a continuous stream, with the decoding operations somehow embedded in the data. The second option was to overhaul the entire broadcast paradigm and replace it with a peer to peer communication model, where the movie player and movie server can both communicate with each other and are both working together as equals to bring the movie data from a source and display it.

The second option was chosen as the direction for O. Although this greatly increased the complexity of the O models and designs, it also opened far more possibilities for other multimedia issues, such as interactivity, dynamic resource management, and context-sensitivity. This choice was more appropriate for a research tool aimed at exploring the future of multimedia.

So, given that O is based on equality of movie server and movie player, the question is how are the tasks of server and player assigned? After all, the OStreams which run the server and player are both identical in function. Where is the difference made?

The answer is that all of these tasks are divided and assigned by the movie objects themselves. In the O model, a single movie object is expressed as a module group. This module group is itself divided into different groups which read data from disk, encode data, decode data, display images, handle interaction, etc. Of course, some of these groups will belong on the movie server machine, and some of these groups will belong on the movie player machine. At first, however, all of these groups will be a part of the same module group which is running on a single machine. So the issue is how to split that module group to operate across multiple machines.

O provides a mechanism for module groups to be sent to and handled by remote machines, while still maintaining their identity as children of their parent module groups, and also maintaining their original connections to the other modules in parent module group (see figure 2.16). In the case of the movie object, the entire object would start off as a single module group running on one machine. The module group would be told where the movie player is. The module group would then send the appropriate module groups (decoder, displayer, etc.) to the movie player. It could even send them to multiple movie players (for example, if the video hardware and the audio hardware were on two different machines). Once distributed yet still fully connected, the movie object could then begin the flow of data through its modules, thereby bringing the movie data from the disk, through the network, to the viewer's screen.

This is the most important part of O, and to understand it requires understanding of both the basic underlying support for transportable objects, and how those support functions are made available to the typical O object.

2.5.1 OStreams as servers

Transportable objects in O are made possible through the design of OStreams as full-fledged servers. As mentioned before, OStreams are the means by which OModules are interpreted and run. Technically, OStreams are actually OModule servers. Normally, they are just used to run a single module. However, remote clients (such as other OStreams) may connect to the OStream and direct it to handle other module groups. These clients may also send messages to modules being handled by the OStream, and the clients may direct the OStream to send the outputs of a module to some remote

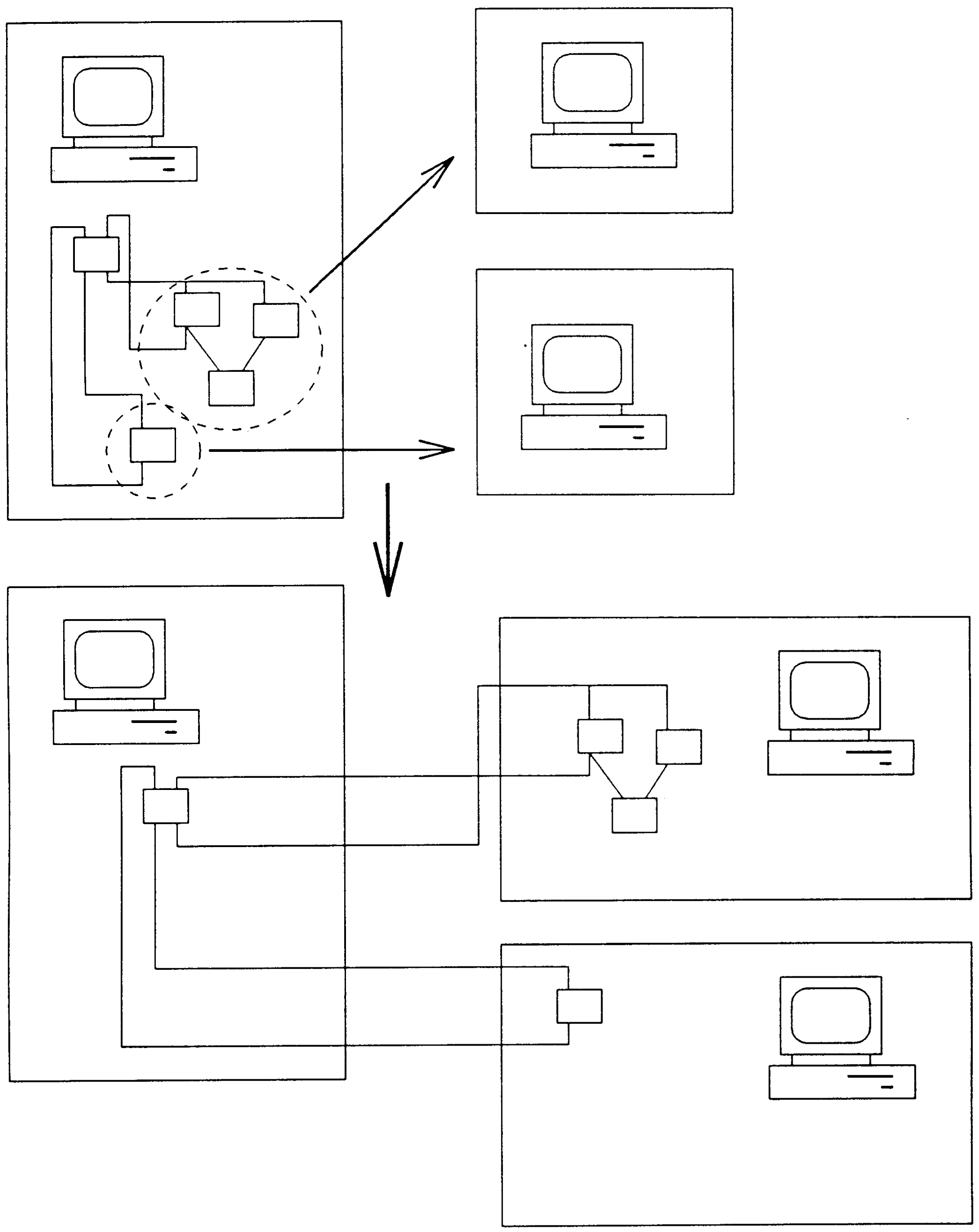

Figure 2.16: OModules being transported

site.

With these functions available, it is possible for a module group to "send" one of its child modules to a remote OStream, yet still have the child module act as if it were still connected to the other children in the module group. This is exactly the paradigm that was required for the movie object model in O.

The final point about OStreams is how they are named. If a movie object is going to be told to send its child modules to a particular OStream, someone must figure out the name of the OStream and communicate it to the movie object. Since OStreams are basically network servers, they can be named in the tradition of all network servers - by a hostname and a port number. This pair is sufficient to uniquely identify any OStream running anywhere on the network.

2.5.2 Modsend OModule class

Given a set of modules that a module group wishes to send to a remote OStream, and also given the hostname and port of that OStream, all that is needed is a trigger to cause the modules to be sent.

A special built-in module class, called a **modsend** module, is used to perform this task. The configuration information for this module is a list of modules that are to be sent to a remote site. The modsend module and the modules that are going to be sent must all belong to the same module group.

The modsend module is triggered by sending it a message in the form (<hostname> <port>). This will establish a connection with the remote OStream, send the named modules to the OStream, and turn the named modules into aliases so that messages to those modules get routed back and forth to the remote OStream.

When all of this rearrangement is complete, the original module group will still be a single coherent module group, but it will span across two OStreams. The original OStream still contains all of the modules it originally had, but some of those modules will simply be placeholders, or aliases, to the modules which have been sent to the remote OStream. On the remote OStream, only those modules which were sent there will be running there - the remote OStream does NOT get a full copy of the entire module group.

In the current implementation of O, not all modules may be sent across the network. Only module groups may be sent. This is not a problem, since a module group may be wrapped around any single module that needs to be sent, and in any case only large functional blocks should be sent to a remote site, which usually implies module groups.

2.5.3 Delayed instantiation

Delayed instantiation and module sending are two closely related concepts. The reason for this is efficiency. When a module group is instantiated, memory is required to keep track of all the modules in that group, and all the modules in the groups under it, etc. When that module group is sent over the network, all of those modules have to be sent over too.

If it is known, however, that the module group will not be used until it has been sent over the network, then there is no point in instantiating the module until it has been sent. This is where delayed instantiation is used. A module group may specify that certain child modules should not be instantiated immediately. This is done through the **delayinst** built-in module. The configuration information for this module is a listing of the modules whose instantiation will be delayed. The delayinst module and the modules it lists must belong to the same module group.

The modules listed in the delayinst module remain uninstantiated until one of two things happens. The first is that the delayinst module gets a message, which causes all the modules it lists be instantiated. The second is that one of those modules is sent to a remote OStream, in which case the module is immediately instantiated on the remote OStream, and on the local OStream the module is instantiated as an alias.

In most O objects, the modsend module and the delayinst module will be used in pairs.

2.5.4 OStream forking

When several modules have been sent to the same OStream, that OStream might start getting bogged down. Other problems might occur if badly written O objects forget to clean up themselves, or worse yet if they cause the OStream to crash, taking all of its modules with it. An example of where this might occur is an OStream which is constantly running as a movie player. O objects will occasionally be sending modules to this OStream, and because things are never perfect, something is eventually bound to go wrong. Perhaps a memory leak, perhaps one module hogging all the resources of the OStream, whatever. In any case, the robustness of the OStream movie player is too closely tied to the robustness of the movie objects it is playing.

To address this problem, OStreams are given the ability to *fork* themselves. When an OStream has forked itself, a new, completely separate OStream is created. Under UNIX, this OStream will have its own memory space, its own resources, and its own slot of processing time (see figure 2.17). Thus, if something goes wrong with the new OStream, the original OStream is not hurt or crippled, it simply loses a child OStream.

Since OStreams are also servers, they may be directed to fork themselves by outside processes. This is the purpose of the **fork** built-in module. This module takes in the (<hostname> <port>) name of an OStream (local or remote). The module then connects with that OStream and directs it to fork itself. The newly created OStream will have its own new (<hostname> <port>) name, which is communicated back to the fork module and the module sends the new name out its output. This new OStream name is suitable for use by, say, a modsend module. It is not unusual to see modsend, delayinst, and fork modules together in a combination, and in fact this combination is required by the Host/OStream model described in the next section.

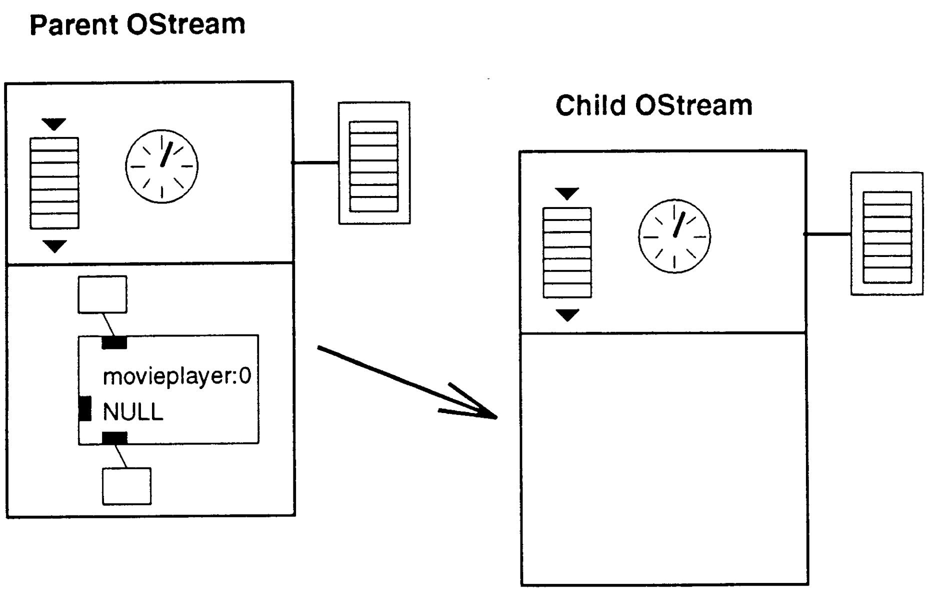

Figure 2.17: OStream forking

2.6 The Host/OStream model

OStream forking, module sending, delayed instantiation, and remote module handling are powerful features of O, which can lead to a disorganized disaster if used without policy.

After studying many of the likely applications for O, the Host/OStream model was chosen as the main policy for developing O applications. This policy is not technically a part of O - O does have some support for this model, but does not actually enforce it, and users are still free to use whatever model they wish.

The basis of the Host/OStream model is to set up special Hosts which accept and display O multimedia objects. However, these multimedia objects are not sent directly to a Host. Instead, the Host forks off OStreams which accept the multimedia objects as they are sent. These child OStreams then run whatever decoders are necessary on the data, and pass the final result back up to the Host, which takes care of display. The users also interact directly with the host, which then relays the information to the child OStreams. These OStreams then communicate this information to the original multimedia objects so they can react accordingly.

As far as implementation goes, these Hosts are actually OStreams themselves, probably running some module group which handles the details of display and interaction. Each Host might be able to display an entire multimedia presentation, or it might just concentrate on one type of media (video, sound, text, etc.). By making Hosts the same as regular module groups and OStreams, the current O model fits perfectly into the Host/OStream model. The only thing that is missing is a means for a Host to communicate with one of its child OStreams. This is explained later.

2.6.1 Host/OStream isolation

This elaborate multi-level system may seem excessive, but it is designed this way for a good reason.

In electronics, when two circuits need to be interfaced together, the circuits are often linked through an isolator device. This device allows information to flow through it, but keeps the inputs and outputs electrically isolated, usually by converting the information to and from some non-electrical representation such as light. Engineers use an isolator in a situation where one circuit might misbehave, and if that circuit blows up, the isolator will keep it from taking the other circuit with it.

O is faced with the same situation. Hosts are generally supposed to stay around for a while, able to play many different multimedia objects at different times. However, the multimedia objects that will be coming over to the Hosts may be unstable or badly written. Rather than directly "wiring-in" the multimedia objects and allowing them to send modules to the Host, the Host instead spawns off a more expendable OStream, which can take in the modules and communicate back with the Host through the equivalent of an isolated communication channel.

2.6.2 Host/OStream communication

So far, all communication in O has taken place with messages through connections. However, as described earlier, a Host and OStream need to communicate through an isolated channel, which means that a direct connection will not be acceptable. Instead, there is a special pair of modules for sending and receiving messages across the isolated gap between Host and OStream.

Toparent/Tochild OModules

A child OStream may send messages to its parent OStream (the OStream which forked off the child) using a **toparent** module. This communication is different from a direct O connection, because if something goes wrong with the child OStream, the parent will not be affected. Any messages that the child wants sent to the parent are simply sent into the input of the **toparent** module.

The parent OStream receives messages from the child through a **tochild** module. Any messages that a child sends into the input of a **toparent** module will appear at the outputs of the parent's **tochild** module. The converse is also true - messages that a parent sends into a **tochild** module will appear at the outputs of the child's **toparent** module.

Figure 2.18 shows a Host/OStream movie object and player in operation. Note the use of the **tochild** and **toparent** modules to communicate between the movie object (which is split across server and client machines) and the movie player.

The only major difference between **toparent** and **tochild** modules is that a child always has exactly one parent, but a parent may have multiple children. Every time a child is forked off from an OStream, it is assigned a unique number. When a message from a child appears at a **toparent** module, it is accompanied by the number of the child which sent the message. Conversely, when a message is sent into a **tochild** module, it must be accompanied by the number of the child that is to receive the message.

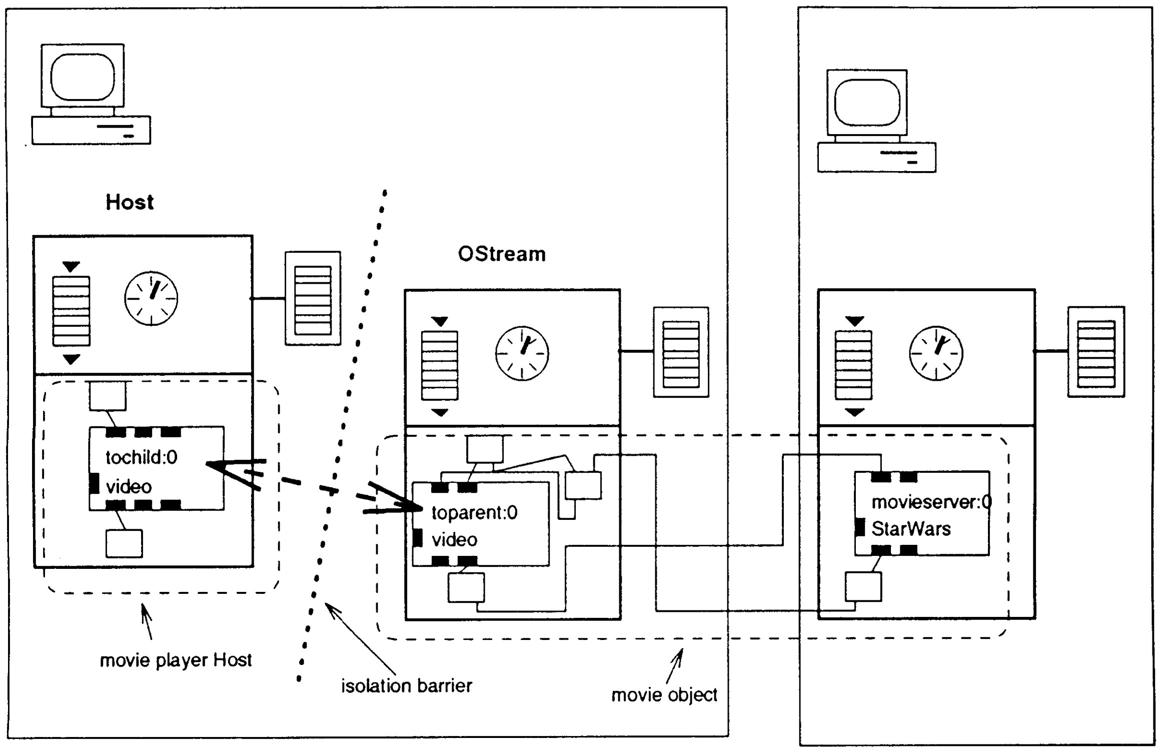

Figure 2.18: Host/OStream model

To help an OStream keep track of its children, the **newchild** module notifies the OStream every time it has forked off a child, by sending out a message containing the child's number.

Resources

Sometimes an OStream and its child may wish to communicate through several separate channels. For example, the child may be sending video information through one channel, sound information through another channel, and the parent may be sending user feedback through another channel. Using only a single pair of **tochild** and **toparent** modules makes this division awkward to implement.

Instead, each **tochild** and **toparent** module communicates over a single channel. These channels are given names, called *resources*. The **tochild** and **toparent** modules specify the resource they will use as their configuration information. **Toparent** modules will only communicate with **tochild** modules that have the same resource name for their configuration information, and vice-versa.

So, most Hosts will have several **tochild** modules which they will use to take in data for their various output modes (video, audio, etc.), or to send back data for different kinds of user feedback.

The reason the different channels are called resources is that the child is usually making a resource request by sending a message to its parent. Sending video information to the parent, for example, requires the parent to use resources for displaying that video information. Different channels of communication will generally require different sets of resources, so even though the mapping is not strict, the name still stuck.

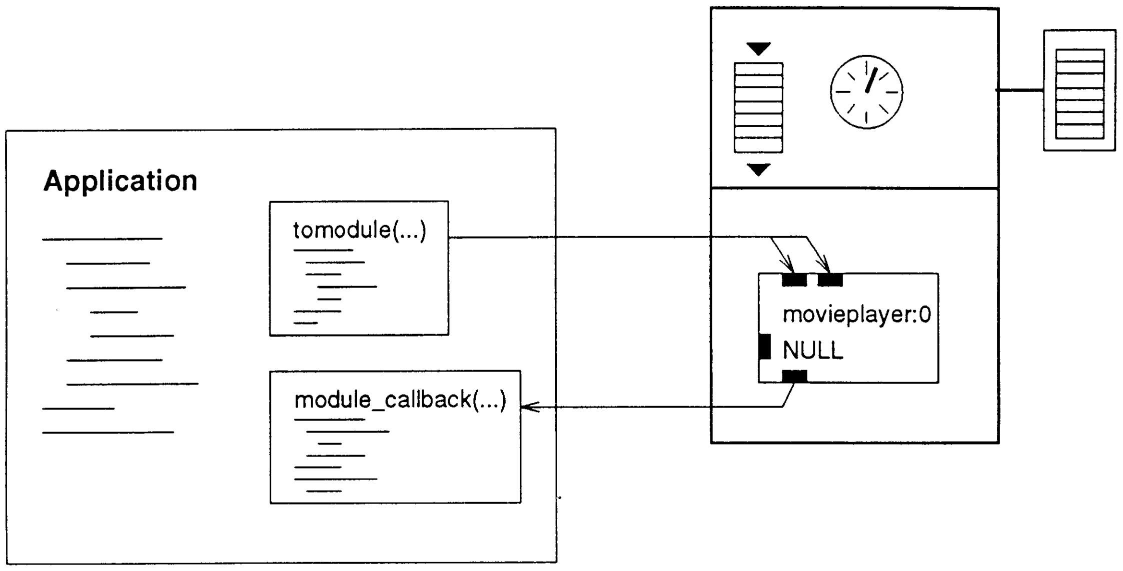

Figure 2.19: Application interface

2.7 Applications Interface

The final issue in O is the interface that applications will use to access the functions of O. The interface should be simple and straightforward and should require little if any knowledge of O, but it should still allow the full power of O to be used.

The solution is to give the application an interface to a single O module. The application will be given functions which it can use to send messages into the inputs of that module. The application will also be able to register callbacks which will be called when the module sends out messages (see figure 2.19). This way, the application can easily wrap itself around the entire module. All the application needs to know is how to manipulate the Dtypes that are sent to the module and received by the module. Everything else is hidden from the application and handled by O.

Once the application has an interface to a single O module, the application suddenly has access to all of the functionality in O, because that single module may be an entire module group, or a remote alias module, or whatever. As long as the application has ties into that single module, the application has ties into everything.

2.7.1 Application as a host

Under the Host/OStream model, applications will generally interface themselves to Host module groups. This way, the application establishes itself as a Host for displaying and interacting with O multimedia objects.

The actual process an application uses to access an O multimedia object is a little backwards. The application must first initiate a request for that object, and tell it the (<hostname> <port>) that the movie should be sent to. Once the request is made, there is an unpredictable amount of time before the O multimedia object actually starts up and connects to the Host, so the Host must always be ready to accept the new multimedia object. A well-written Host module group will take care of all these issues, by allowing the application to ask the Host to find the O multimedia object it wants, then prepare itself to handle that object.

An O widget for X

Since O is primarily designed for the transport and display of multimedia, it would be nice if O could easily fit into existing display systems. An example of this is the X toolkit, which uses object-oriented display elements called **widgets**. An O widget for X would be an especially nice way for programmers to use O knowing hardly anything about how O works.

However, a single O widget is impractical since O is capable of doing far more than can be encapsulated by a single widget. A suitable compromise is to have a whole set of O widgets, each of which handles a different type of interaction or output, such as video, text, sound, or maybe some combination. This would require that the O multimedia objects which are connecting to these widgets be able to reconfigure themselves based on what kind of output the widget will be displaying, which is the purpose of the context-sensitive multimedia model.

2.8 Future directions for O

This chapter has repeatedly emphasized that O is a research tool, and that its purpose is to turn ideas into working processes as quickly as possible. This being the case, design decisions for O almost always favor functionality over efficiency.

O currently has most of the functionality it needs, and O is designed to allow new functionality to be added easily (by writing new built-in modules or creating new objects). The next major step for O is to improve its efficiency and begin to move O into "real-time" applications. One possible goal is to enable O to meet temporal specifications (like Chris Lindblad's *ViewStation* [Lin92]) as well as functional specifications. This will require O to perform temporally sensitive functions, such as maintaining a constant bandwidth data stream over the network, and paying closer attention to resource usage. The efficiency of O's implementation can always be improved, and some of the designs may also need rethinking. For example, the decision to fork off OStreams by using UNIX fork was a quick way to achieve Host/OStream isolation, but that decision may prove to be inefficient as Host/OStream communication becomes more intense.

Of course, O will only improve according to how much it is used. The current plan is to incorporate O into as many applications as possible, and expose any areas in which O is deficient or inefficient so that O can be tempered into a robust and useful tool.

Chapter 3

Context-Sensitive Multimedia

Object-oriented programming will follow a steady progression into the field of multimedia design, and as a result many new applications and interactive models will follow. One possible model is context-sensitive multimedia. This chapter begins with a detailed discussion of object-oriented multimedia, which is the prerequisite for context-sensitive multimedia. The chapter then develops the object-oriented model into a discussion on context-sensitive multimedia.

3.1 Object-Oriented Multimedia

Object-oriented programming is a well-established method for increasing productivity, improving standardizations, and reducing errors. Often-used pieces of code are encapsulated into "black-boxes" which programmers have found that they can share and improve. Because programmers can construct their applications out of highly functional and reliable blocks rather than building everything "from scratch", programmers can create programs more quickly and with fewer errors. Most importantly, though, programmers can describe their implementations on a higher level, making the step from designing board to implementation shorter and less of a mental gear change.

Electronic multimedia, in its current trend, can benefit from many of these advantages. Multimedia for computers is still in its industrial infancy, and as such it suffers from a great deal of "reinventing the wheel". Few, if any, standards are used, so multimedia designers usually end up striking out on their own, especially if their designs can't fit one of the commercial multimedia authoring tools currently on the market.

Part of the problem is that the standards which have been attempted are at much too high of a level - establishing *the* correct video encoding algorithm (JPEG, MPEG, etc.), standardizing *the* best way to link multimedia objects together (hypermedia), etc. While these approaches are fine for bootstrapping electronic multimedia into a widespread market, they are too inflexible to survive the coming onslaught of new technologies, new models, and new visions for multimedia. What is needed is a standard model for computer multimedia that is universal enough to encompass the needs of future designs and applications.

This is where object-oriented multimedia has its place. An object-oriented approach does not command a high-level design policy, but instead makes the lower-levels of the design easier to work with, build on, and less error-prone. Object-oriented multimedia is destined to become an accepted standard of the future, because it offers a great deal of simplification in programming and design, without costing functionality or flexibility.

3.1.1 Features of object-oriented multimedia

An object-oriented approach has some key benefits which are especially appropriate to multimedia.

Common interface

Interface has many meanings when used in conjunction with multimedia programming. The most obvious meaning is the system that allows a user to interact with a program. The second meaning, which is the one used in this section, is a *programmer's* interface. This interface is the set of methods a programmer establishes for other programmers to interact with the functionality in his programs.

One of the major problems with group programming is misunderstandings in this area. If one programmer is unclear about how others should use his code, then his code can at best be considered unstable, at worst unusable. In all cases, each programmer's individual style of interface adds more complexity and lost time to other programmers.

In multimedia, a similar problem is growing. Source material for a multimedia presentation might come in a variety of forms. For example, consider a video clip that a multimedia designer may wish to use. The source might come on videotape, or even film, or it might come already digitized, perhaps encoded in one of several digital video coding formats.

But let's assume that the video has already made it to the digitized stage, and has even come with programs that will decode the video into whatever form the designer wishes. The problem now turns into a programming problem - how to fit those routines into the application the designer is putting together. It may be that the programmer has chosen his own arbitrary methods of interfacing to the program, which the designer will have to learn and test.

Or, the programmer may have chosen a standard object-oriented approach, which brings with it a standard method for interfacing to the code. If this is the case, then the designer can happily consider the video clip to be a single object. All the programmer has to do is specify what methods are available with that object, and the designer knows immediately how to invoke those methods on the object.

Reusable libraries

Current multimedia in non-electronic form is inherently reusable. Video, audio, photos, and film clips may be duplicated and edited into many different applications.

These multimedia objects are quite simple and the means by which they can be reused is physically obvious.

Electronic multimedia, however, is more complicated because it can actually *do* things, rather than just being shown. More specifically, electronic multimedia has the ability to interact, both with the viewer and with the system around it. So, reusing electronic multimedia is a more complicated issue, because not only does its source data need to be duplicated, its *functionality* must also be duplicated.

The object-oriented model handles this by declaring the encapsulation of both data and functions into a single object. This way, when someone receives a multimedia object, all the data and all the functionality of the object are immediately made available. All that is necessary is some standardized method of invoking those functions, and as explained earlier this is also addressed by the object-oriented model.

So, electronic multimedia can hope to see a future of reusable libraries of functional media objects. Just as there are now file photos and file video, there can be file electronic multimedia, where every object pulled out of the file brings along with it all of its own methods for interactivity, adaptability, and even links to other multimedia objects.

Scalability

Although the object-oriented method was presented as a low-level programming paradigm, it can be scaled to work at higher levels. Objects may be built out of objects, and so may inherit the functionality of all the included objects while adding new functionality for the collective group of objects.

For multimedia, this means that an object can represent a single media source, or it can represent a collective multimedia presentation, or it can represent an entire database of presentations. The object-oriented method works the same across all levels.

3.1.2 Applications of object-oriented multimedia

There are many "hot" issues in multimedia research today, and several of them can be addressed in terms of applications of the object-oriented method.

Resource management

Resource management is one of the biggest problems that will affect the distribution of electronic multimedia. Multimedia users are going to have very limited resources at their disposal - computing power, network bandwidth, screen space, memory, etc. Electronic multimedia in its current form consumes tremendous computing resources, and will likely do so even more in the future. If a user attempts to combine two multimedia objects which together use more resources than his system can give, there will be trouble.

Fortunately, increasing attention is being given to *scalable* algorithms, which will result in multimedia objects that can vary the amount of resources they require, in exchange for better or worse quality of the final product. If such scalable multimedia objects are constructed, then systems can have central resource managers, which can mediate between objects and dole out system resources to the different objects as it sees fit.

However, each multimedia object will scale in its own way. A scalable video stream will doubtless have a different method for adjusting resource usage than will a scalable audio stream. If a central resource manager is going to work at all, it must know all of these scaling methods for all objects it will be controlling.

Or, the resource manager can know none of the scaling methods and leave the scaling up to the multimedia objects themselves. An object-oriented design of multimedia objects will make this possible. As long as every multimedia object understands resource allocation requests, the resource manager can work simply by telling each object what resources have been allocated for it and letting the objects reconfigure themselves to work with what has been given them.

Central control

Resource management is a specific example of a more general issue, central control over all the objects in a multimedia system. An example of this is the various "desktop" windowing systems (Macintosh, NeXT, etc.), where the user may globally specify preferences for their applications, usually for things like color, or mouse behavior. A multimedia system may desire similar kinds of central controls, which will affect the behavior of all objects in the system.

The object-oriented method makes this possible by allowing such a central controlling device to communicate with the objects through a common interface without having to know the internal operations of these objects. So, the desktop preferences can simply tell each object that its foreground color is blue, and each object will adjust itself accordingly, if appropriate.

Synchronization

Synchronization is the problem of progressing two or more time-ordered media sequences (such as video and audio) in the same time scale, to make sure that they "go together". While the object-oriented method is not an immediately obvious help, it does provide a good stepping stone to several possible solutions.

One such solution is actually based on MIDI, a real-time electronic musical instrument network, which has been used for performances and in studios for several years. With electronic instruments, there is often the problem of several instruments having their own preprogrammed sequences which they must play together in synchrony (drum machine, sampler, keyboard, etc.), a situation where no one instrument is the master of all the other instruments. In this situation, one instrument is chosen to synchronize the others, and that instrument sends out a synchronizing pulse at regular intervals to the MIDI network. The other instruments pick up these pulses and use them to synchronize themselves to the others. The same synchronization pulse is sent to every machine, and every machine is free to figure out what it will do with those pulses.

A similar scheme is a possible method of synchronization for multimedia systems, where one object sends out synchronization messages to all other synchronized objects. This is most easily accomplished in an object-oriented environment, where each object can receive and understand the same synchronization messages, but act differently depending on how they internally achieve their synchronization. In such an environment, one could imagine objects with a specific synchronization input, which is wired directly to a master synchronization object, so that one object could be synchronized to any other object. Although it is unlikely that objects will have to synchronize themselves in such a flexible manner, it is useful to have such an ability in mind when designing synchronized systems. Otherwise, the system might end up with a dedicated method for synchronizing video and audio, but as soon as some other kind of object comes along, such as a running text commentary or closed captioning, it will be impossible to synchronize with the video and audio simply because it was left out of the design.

Interobject communication

Synchronization is a specific example of a larger issue, which is interobject communication. Many applications will use several multimedia objects, and these objects may have to interact with each other. This is where it becomes crucial that objects rigidly adhere to a standardized programming interface. Consider an example where a movie object might be controlled through a standard "VCR-like" interface (play, stop, pause, <<, >> buttons). Now consider that the same object might have to be controlled automatically to fit into another application such as a slide show, where it receives instructions from other objects as to when it should start playing or pause. Now consider that the original VCR-like interface might be used to control other objects, such as that larger slide show presentation, where play, pause, and rewind might have different meanings to the overall presentation.

In all these cases objects will have to communicate with each other, possibly in some hastily constructed arrangements. It is vital that these objects can communicate in a standardized fashion, rather than devising a new ad hoc scheme for each application an object will be sent into. Here, not only is the object-oriented method necessary, but a carefully considered design for the functionality and control of each object is also required. If this is done right, then making multimedia objects work together should be as easy as snapping together LEGO bricks, or at worst wiring up a stereo system. Every object should be expected to at least recognize the common messages that other objects or an application might want to set out.

3.1.3 Object-oriented multimedia and O

There are many ways of establishing an object-oriented system for multimedia. O is one such system. O is actually a research testbed for exploring the possible applications of object-oriented multimedia, and as such it will probably not be the multimedia system of the future. However, O does have at least a minimal set of the necessary ingredients for an object-oriented multimedia system, concentrating especially on the features mentioned before: common interface, reusable libraries, and scalability.

Common interface

Different object oriented systems have different methods of communicating with their objects. A programming language such as C++ [ES90] invokes object methods in a manner similar to function calls. O uses message passing, a more dynamic, process-oriented approach.

In O, every multimedia object is an OModule, figuratively a "black box" with inputs and outputs. Objects communicate with each other by passing messages from outputs to inputs. Applications communicate with objects in the same way, by passing messages into the inputs of an OModule and receiving messages sent from the outputs of an OModule. This is the common interface which is supported by O.

As mentioned before, this common interface only serves to standardize the means by which objects or applications invoke the functionality of an object. O does not establish any minimal set of functionality that objects should have, nor does it establish a policy for object-oriented multimedia (except the Host/OStream model, which is only supported, not imposed, by O).

Reusable libraries

O is heavily based on the concept of reusable objects. Objects in O are machine-independent, meaning that they can be used identically on all machine architectures. In addition, O makes it especially easy for objects to be included within objects, which makes it easy for an object to be reused and incorporated into a larger object or an application.

However, O's strongest point for reusable libraries is its object location system. A great deal of O is spent on real-time location of objects, both as files or as remote entities. In order for an object to be used, one only needs to specify its name, and at runtime the O system will go out and find that object. In programming, this is known as *dynamic linking*, except that this process can go beyond the confines of the machine and extend its search to remote sites, all invisibly to the user. This feature of O insures that not only can objects be reused, but they can also be found easily in order to be reused.

Scalability

O bases most of its design on the principle that every object is an OModule. The most basic, low-level block of functionality in a system is an OModule, and a full multimedia presentation or even an application is also an OModule. This insures that the object-oriented scheme for O is scalable across all levels of design.

Among other things, this makes the design process much simpler. A design can be sketched out in a very high-level form using large function blocks. Such a design can quickly be expressed in the form of an O network. Then when the functional blocks have to be designed, the exact same process can be used. The design is now occurring on a different level, but the mechanics of the design process and its expression in O remain exactly the same.

3.2 Context-sensitive multimedia

Context-sensitive multimedia is an experimental design for multimedia objects, which puts a subtle twist on the object-oriented programming model. Traditional object-oriented programming assumes that objects are static entities which fit easily into applications because they have programming interfaces which are easy to use. Context-sensitive multimedia makes less of a distinction between the application and the objects. A context-sensitive multimedia object is an active, autonomous process, which is placed into an environment, or *context*, that is arranged by the application. Metaphorically, the application is a director who sets the stage for the objects, the objects are the players which interact with each other, with the set, and with the director (thanks to Brenda Laurel [Lau91] for this analogy).

Under this model, the application has less direct control over the objects it is using. Instead, the application communicates with the objects as an equal, as if the application were just another object operating in the environment. Of course, the application is a special object, because it has direct control over the environment, and thus has indirect control over the objects in the environment.

So why sacrifice an application's direct control over its objects? What benefits can be expected from the context-sensitive model that will justify such a sacrifice? And under what circumstances is the context-sensitive model an appropriate choice for an application?

3.3 Applications of context-sensitive multimedia

Context-sensitive multimedia is not a universally acceptable model for all multimedia applications. As mentioned before, the model demands a sacrifice in control from the application, and this is not always appropriate.

Context-sensitive multimedia was originally proposed as a model for large *multimedia presentations* that come packaged as a single multimedia object. For example, when an application invokes such an object, the application might receive several video and audio streams, closed captioning text, and references to other relevant object. The object may even contain several different versions of its content, each directed to a different viewing audience.

Under the pure vanilla object-oriented model, such a presentation would need quite an intelligent application to drive it. The application would need to understand all the things the presentation could offer, decide which of those elements are useful, and direct the presentation in displaying those elements. The object-oriented model simplifies the communication between the application and object and hides many gritty details from the application. However, the application still requires a fundamental understanding of the presentation's capabilities before it can use the presentation. The major consequence of this fact is that the complexity of multimedia applications will have to scale according to the complexity of the presentations they will run.

The context-sensitive model is an attempt to shift this balance, by placing a greater burden of complexity on the context-sensitive objects, while allowing the applications to be as simple or as complex as desired. More complex applications will have the benefit of finer control over the presentations they display.

According to the basic context-sensitive multimedia model, an application waits for a multimedia presentation object to arrive. The application then tells the object as much as it can about the *context* that the object will be operating in: how much screen space is available, if there are any facilities for playing sound, color capabilities, and even information about the viewer's interests and preferences. The presentation object then sorts through all this information for things that it finds relevant, and uses that information to configure and adapt itself to the environment into which it will be placed. At that point the presentation can begin, and the object will start sending data to the application, which will presumably know what to do with it. The object must also be prepared to reconfigure itself in the middle of a presentation, since the context may change due to user intervention or a reshuffling of resources. Once again, how smoothly this change is communicated to the object is a function of the complexity of the application.

In order to adapt themselves to multiple environments and display platforms, multimedia objects are going to have to be much more complicated and intelligent. This means that a great deal of time will have to be spent on the production of multimedia objects, which is a reasonable expectation for future electronic media that will be published or sold on a large scale.

3.4 Context

Context is defined as a region of resources allocated for a presentation. The most obvious and universal resources are space and time. Space refers to the visual (and aural) area in which the presentation will occur, most likely on a display monitor. Time refers to the expectation that a presentation will have a beginning and an end, between which the presentation will occur. Other resources depend on the application, the system the application is running on, and even the audience which is viewing the presentation (the viewer's attention can be modeled as a resource).

Content is another property of contexts. A context can have a specific bias to its content, which may be inherent to the application (a liberal versus a conservative newspaper), or it might depend on the preferences of the viewer. In either case, the content of multimedia objects can also be constrained and directed to fit into a context.

When an application and a multimedia object first interact, the application must communicate all of these resource constraints and preferences to the object. To organize this communication, the preferences have been divided into two categories, **contextual modes** and **content cues**.

3.4.1 Contextual modes

One of the most important issues that a context and an object must resolve is what forms of media are going to be used. The different forms of media (video, audio, text, etc.) supported by a particular context are called the **contextual modes** of the context. The first negotiation between an object and a context involves the context telling the object what modes are supported or expected from the object. The object then activates those modes within itself (if they are available). The context continues the negotiation by supplying parameters about each mode. For example, if one of the modes is a video stream, the context might indicate the amount of area that the video must fill, whether color is available, and what form the data should be converted to before being passed into the context.

In some cases the object may wish to negotiate back with the context. If, for example, the allotted screen space is too small or the context is demanding its data in a form that the object doesn't know about, the object may wish to tell this to the context in the hope that the context will return with more favorable requests. Once again, how well this is handled is entirely up to the context. The context might be able to accommodate negotiation on many points, or it might simply ignore all such requests from the object. The object should be able to continue operating in either case.

In the O model for context-sensitive multimedia, each contextual mode is given its own *resource*, or channel of communication. Upon startup, these channels carry the configuring information for the different contextual modes. While the object is running, these channels also carry the actual data that the object supplies to the context, and carry runtime messages back from the context to the object. A special channel, called the **control** resource, is used to start the whole process by telling the object which modes are expected.

3.4.2 Content cues

Through the contextual modes, a context can determine the *form* of a presentation. To influence the *content* of the presentation, the context uses a channel called **content cues**. This channel can carry a wide range of information, anywhere from a desired political slant to an entire personality profile of the viewer.

Consider for example a multimedia newspaper application. The application may be made from several different contexts, each of which represents a "sports page" or an "international page", etc. The newspaper has access to a large database of multimedia "articles", each of which is capable of displaying itself in several different formats (a text article, a photo, a map, etc.). Each article is also capable of rewriting itself for different uses, since the same information will be expressed differently if written for, say, a conservative newspaper rather than a liberal newspaper.

Now the newspaper begins to put together its articles. Somehow, the newspaper has been given a set of today's articles, which are multimedia objects in the database. Each context, or "page", goes and gets its articles from the database. Since these articles are context-sensitive, or able to adapt themselves to their application, the context must tell the articles just what is expected of them. If the sports section is supposed to be heavily photo-based, then the context will negotiate with the objects until about half of them agrees to send photos while the other half agrees to send text articles. This is how the context communicates its contextual modes.

The context can then fine-tune the content of the articles. For example, one sports page context can declare that it is oriented more towards statistics than stories. That context would tell its articles to rewrite themselves to include more statistics. Or the political page context might know that it is part of a conservative paper, and tell its articles to be rewritten[1] from a conservative viewpoint. These content instructions that the context passes to the articles are called "content cues".

The content cues are expressed and communicated from context to object by a special *knowledge representation* being developed by Jon Orwant. This knowledge representation is used by a program called **editor** (also by Jon Orwant), which can search through a database for information that best fits the description specified in this knowledge representation. Context-sensitive objects can use a similar technique to take content cues and internally search for the best way to fit an article to those content cues.

Content cues can be even more powerful if they are combined with the preferences of the user. For example, an article may be written from a different viewpoint if it is being read by a man or a woman. This is possible if the application is combined with a user-modeling system, such as Doppelgänger, by Jon Orwant [Orw91]. This system maintains user models of several people, where each model is expressed using the same knowledge representation that the content cues use. If the viewer's user model is included in Doppelgänger's database, then the content cues already built into a context can be improved by the viewer's user model, and the whole thing can be sent to the object as one large set of content cues. This will have the effect of adapting the multimedia objects to fit not only the needs of the application, but also the interests and preferences of the viewer.

This system can also return the favor to Doppelgänger by telling Doppelgänger what the user's response to the presentation was. One of Doppelgänger's basic tenets is that user models are dynamic, so if Doppelgänger made a bad content choice for a viewer, it wants to know about it so it can update its internal model for the user and make a better choice next time.

[1]Note that "rewriting" an article can be done many ways: it might actually be rewritten by a very talented computer, or the computer might just find an appropriate article in the database that has already been written.

3.5 Contextual hierarchy

Many applications will have more than one context which will need to share common traits, and a system may contain many applications which will also need to share common traits. *Contextual hierarchy* is the method by which contexts may inherit traits from each other. Each context has one parent and any number of children in the hierarchy. There is also a root context for the entire system which has no parent. The contextual hierarchy ignores the boundaries of separate applications, so contexts from several different applications may inherit from the same context.

There are several possible applications for a contextual hierarchy.

3.5.1 System capabilities and preferences

Often there is information that will be common to all contexts on a given system. For example, the hardware capabilities of a system is something that all contexts in the system should be aware of. This sort of information would come built into the root context of a system, so the information will be inherited by all contexts in the system. Another piece of common information is something which many desktop environments call "user preferences". This usually refers to adjustable details about the user interface. If a user-modeling system, such as Doppelgänger is being used, then the root context can also contain the user model of the whomever is using the system, and pass that information on to all contexts in the system, so that all contexts will automatically tailor themselves to the interests of the user.

3.5.2 Resource manager

System resources are something that all contexts should be aware of, but the resources should be controlled from a central point. This is once again a function of the root context. A central system resource manager can run separate from all other applications and make its decisions known through the root context, where they will be passed to all contexts in the system. Note that this requires the contextual hierarchy to be dynamic - changes at one level of the hierarchy must immediately be propagated to the lower levels.

3.5.3 Common interface

A hierarchy of contexts allows an application's interface to be designed hierarchically. An application might have its own interface structure, while each context within the application might have an additional interface. For example, the application as a whole might have "play" and "pause" buttons, but the audio context of the application would have a volume control, while the video context would have a brightness control. Within both of these contexts, the "play" and "pause" buttons for the application would still work. This could be done using a contextual hierarchy. The application would have its own context with the "play" and "pause" buttons, while the video and audio with their volume and brightness controls would be child contexts of the application. Within the video and audio contexts, only the appropriate volume or brightness control would be active, but both contexts inherit the functionality of the "play" and "pause" buttons from the application's context.

3.6 Context-sensitive multimedia and O

There are many possible ways of implementing a system to support context-sensitive multimedia. This section describes one possible way, which uses O in its capacity as a research tool for object-oriented multimedia. Context-sensitive multimedia, while not directly related to object-oriented multimedia, is very easily implemented on an object-oriented multimedia system such as O.

Note that O is not in itself a context-sensitive multimedia system. What will be described in this section is a model for using O to build multimedia objects in a context-sensitive manner, and to build applications that are based on contexts.

3.6.1 The Host/OStream model

The Host/OStream model was introduced in the O chapter as a policy for developing O applications. This model will be used as the basis for a context-sensitive multimedia system under O.

Under the Host/OStream model, an application has several Hosts to which multimedia objects attach themselves. Multimedia objects are actually spread out across several machines, but originate from a single machine, usually the machine where all the source data is kept. When an object is sent to a Host, the object is distributed such that part of the object stays on the original machine while part of the object is sent to the Host.

An OStream represents an object's connection with a Host. In order to maximize the robustness of an application and its Hosts, the OStreams are isolated from the Hosts. OStreams and Hosts communicate with each other through special channels called **resources**.

In general, an object is responsible for acquiring, transporting, and decoding multimedia data. The Host is responsible for displaying that data, and maintaining the application's resources for displaying data. The Host uses the **resources** to tell the OStreams what kind of data it expects from them, and the OStreams use the **resources** to pass that data up to the Hosts.

3.6.2 Anatomy of a context-sensitive object

This section describes how one might build a simple context-sensitive object in O. A simple context-sensitive object must meet the following criteria:

- The object represents a single time-ordered sequence. This includes, but is not limited to, a sequence of video, or audio, closed captioning, or some combination thereof. The sequence may be divided into several parts happening in serial or parallel (i.e., two video streams at the same time), but any sequences happening

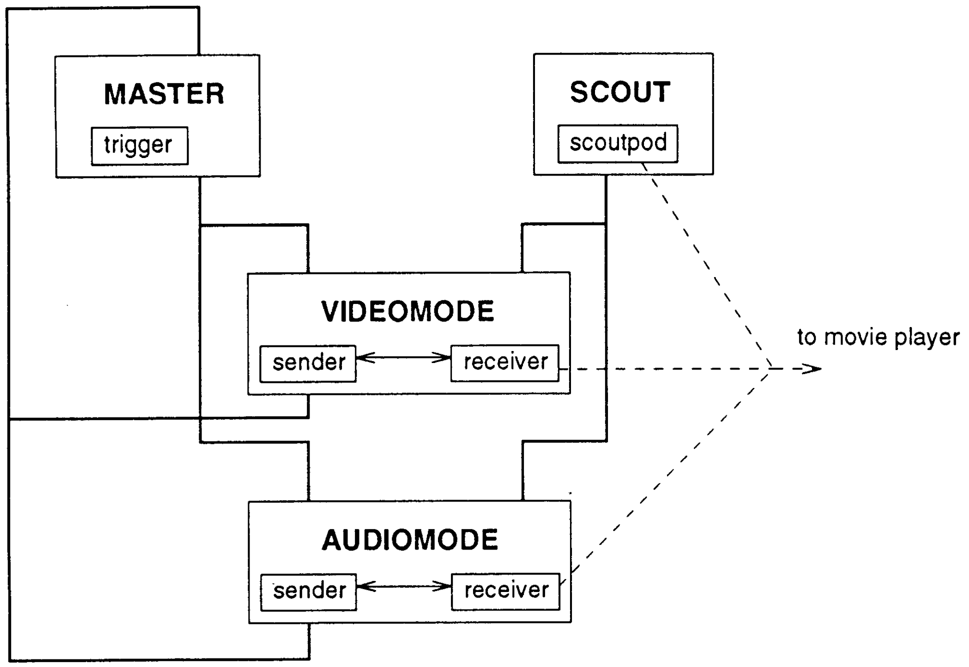

Figure 3.1: Context-sensitive movie object

in parallel will be forced to progress together, not separately.

- The object contains one or more **modes**, which represent the different forms of media the object can display. This includes video, audio, text, etc.

- The object can be driven or synchronized from a single master clock source.

Given these criteria, the object can be constructed as a single module group, consisting of three different types of modules: a single MASTER module, a single SCOUT module, and one or more MODE modules (see figure 3.1).

None of these modules actually gets sent across the network to the Host/context. Instead, these modules are module groups which contain the modules that get sent over. The following is a detailed analysis of each module:

MASTER module

This module is the heart of the object, containing only the master clock source from which all the **modes** will be driven. The master module is designed around the

trigger OModule, which is a special module that can keep time accurately and trigger events at regular intervals.

Sometimes the different modes will have to communicate with each other. The master module handles this communication, usually by providing a means for each mode to broadcast a message to all the other modes.

The master module can be controlled by sending it messages very similar to a VCR control, such as play, stop, pause, reverse, forward, fast, and slow. Of course, each object may have its own set of special commands for the master module, such as commands which cause the timer to jump to the next scene rather than the next frame.

In the current model, the master module remains completely on the source machine and does not get distributed to the context.

SCOUT module

This is the first module that makes contact with a new context. When the object is sent to a new context, the scout goes first, and reports back which contextual modes are going to be needed, and also reports any content cues. Any negotiation with the context is carried out through the scout. The scout is also told when the object is going to be moved to a new context, or dropped from a context altogether.

The scout module is a lightweight module which contains only a **scoutpod** module, and a mechanism for sending the scoutpod module across the network to the context which is waiting for it. The scoutpod communicates with the context through two resources. The first resource, **control**, is how the scoutpod learns which modes the context is expecting. This information is sent directly back to the object where it is relayed to all the modes. This resource is also how the context informs the object that it is to disconnect itself from the context, possibly to be redirected to a different context. Upon receiving such a message, the scoutpod causes all the modes to disconnect themselves, then the scoutpod sends itself to the new context to which it has been redirected.

The second resource, **content**, is how the scoutpod receives the content cues from the context. These content cues are passed directly to the mode modules for them to analyze as they see fit.

MODE modules

This is actually a set of modules, one for each mode, called videomode, audiomode, etc. Each mode is wired the same way into the scout and the master modules. The master module transmits its clock signals to all of the mode modules. The scout module also transmits its contextual modes and content cues to all the modes. It is up to the modes to turn themselves on or off depending if they will be needed or not. Each mode is responsible for acquiring, transporting, and decoding the information associated with its media type. Each mode also handles user response from the application, and may send some of those responses back to the master module to affect the master clock.

The mode modules are where the actual work of the object gets done. Each mode module contains two modules: a **sender** and a **receiver**. If a context requires that a certain mode be present, then that mode sends its receiver module over to the context. When the master clock begins ticking, the sender module will start acquiring data from wherever its source is (disk, camera, etc.) and send that data to the receiver. The receiver has decoders built into it which convert the data into whatever form is required by the context.

The receiver module of a mode communicates with the context through a specific resource. For example, a frame-based video mode will communicate with the context through the **frame** resource. The context sends messages through the **frame** resource indicating what form it wants its data in, what size it expects for the data, what color scheme should be used, etc. The receiver module then sends the actual frames up to the context through the **frame** resource, where they are displayed.

3.6.3 Context-sensitive multimedia and X

As mentioned in a previous chapter, X will probably be the first display platform for O, and one of the first steps to making O easily accessible is to make an O widget for X. The context-sensitive model is perfect for this application. To work with this model, the O widget must contain a host which works with context-sensitive objects. This widget may display any kind of medium: video, still, text, whatever. When an object is attached to the host, the host then tells the object what mode it can display. If the object is written in a context-sensitive manner, then it will configure itself to do what the O widget expects from it, and everything will work just fine.

Chapter 4

Demonstrations

This chapter describes two applications which use O to demonstrate the concepts of object-oriented multimedia and context-sensitive multimedia.

4.1 The Bush Demo

The object-oriented features of O were demonstrated in an application based on President Bush's 1992 State of the Union Address (see figure 4.1). The video and audio from the broadcast of the address were broken into 33 "sound-bite" segments, ranging from 10-30 seconds in length. The segmentation is based on content, each segment containing only a single topic. The segments are turned into separate movie objects, containing encoded video, an ASCII transcript of the audio, extracted keywords, editorial annotation, and a high quality "salient" still image, representative of the entire segment.

The application cycles through the objects by requesting and playing them in the order of the original presentation. Each of the movie objects understands simple decoding messages such as requests to change size, or to change from color to black and white. Each object responds to the application's issuing of a pause command by sending over a the representative still image. When play is resumed, the object continues

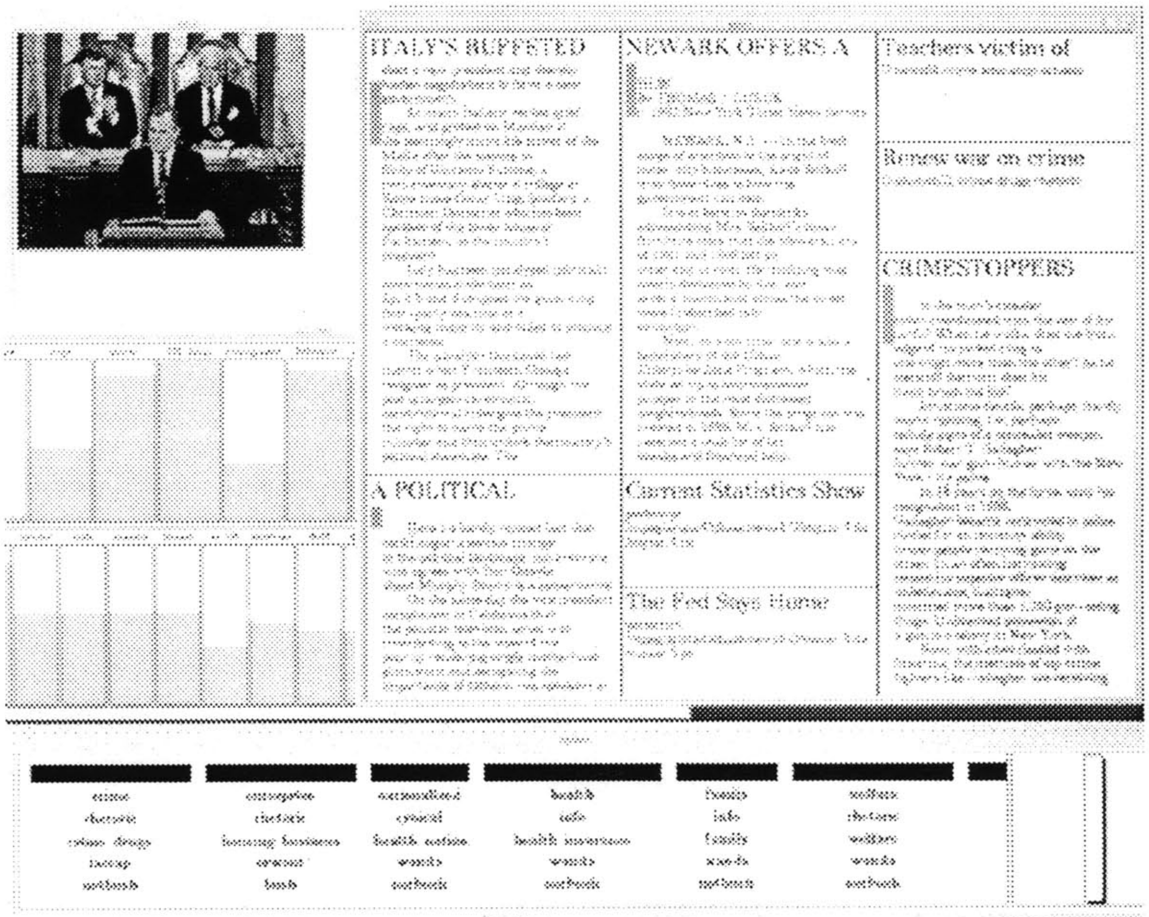

Figure 4.1: Object-oriented State of the Union

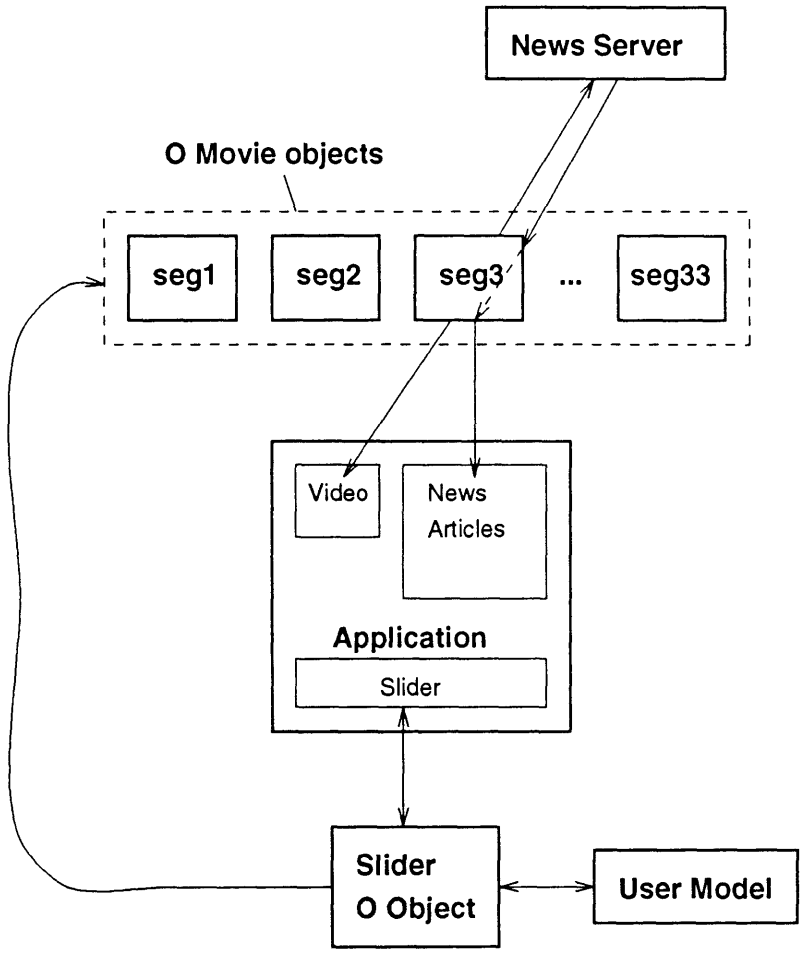

Figure 4.2: The Bush demo system

the movie. The response of the objects to the application's changing requirements, and the system's changing resources, is embedded within them. The application itself need not concern itself with either the segmentation or encoding of the objects.

In addition to being able to respond to the current display context, each movie object is also aware of its own content, and is able to "advertise" that content to other applications. For example, the objects are able to connect to an electronic news server to get current news articles relevant to what the president is saying. These articles are obtained solely through the "efforts" of the movie objects - the application doesn't even have to know about a news server.

The implementation includes a "content-slider" which allows the viewer to sequence through the presentation. This slider is also an O object. It has access to the content information conatined in each of the movie objects. It also has access to models of several possible users. Consequently, the slider can match the movie objects up to the people who would be most interested in them. As a result, the slider allows a user to "fast-forward" to the next segment that that user would personally find interesting. This fast-forward by content rather than by frames is invisible to the application - it is all handled by the methods of the slider object.

4.2 The Interactive Hallways Demo

The first application of the context-sensitive model was assembled by the author as a demonstration of the MASTER/SCOUT/MODE model. The demonstration is a basic application of the concepts in the context-sensitive model. Some elements of the model have been left out, namely content cues and contextual hierarchy.

The demonstration is based on an interactive movie map of the hallways of M.I.T., similar to *Interactive Hallways* of 1978. In this version, the hallways have two *modes* of display. The first mode contains digitized video taken from a camera. The camera was mounted on a Steadicam JR, the cameraman mounted on inline skates. The digitized video is encoded using a special combination of subband coding and vector quantization. The mode contains a scalable decoder which allows the video to be decoded to a variety of sizes, with or without color. The second mode of display is through an overhead map, with a "you are here" marker indicating the position of the viewer as he rockets virtually through the hallowed halls of M.I.T. The maps were obtained from the MIT Office of Facility and Management Systems in CadView form, then converted through AutoCad into a special form used in O. The data for the "videomode" and the "mapmode", the video decoder, and the interaction behaviors (play, stop, reverse, etc.) were all combined and encapsulated into context-sensitive movie objects.

The application consists of two contexts: one can display video, the other can display maps. When a movie object first hooks up with the application, the object is directed into the video context. There the SCOUT module learns that the context is expecting video information, so the movie object activates the "videomode", sends its decoder over to the context, and starts sending video frames to the context. The context also includes a panel of VCR controls which direct the movie to stop, start, reverse, etc. To demonstrate the scalable decoder, the context contains an additional control panel which adjusts the size and color of the video.

The video context also contains a button that disconnects the movie object from the video context and sends it over to the map context. There the SCOUT learns that map information is desired, so the "mapmode" is activated. A diagram of the appropriate map is sent to the context, and the movie object begins sending virtual coordinates of the "you are here" marker. The VCR controls still work in this context, but there is no panel for video size and color - instead, there are controls which allow the viewer to zoom in on portions of the map. Like the video context, there is also a button which disconnects the movie object and sends it back to the video context. Figure 4.3 shows the demonstration in action. Figure 4.4 shows a schematic diagram of the O module networks used to implement one of the movie objects.

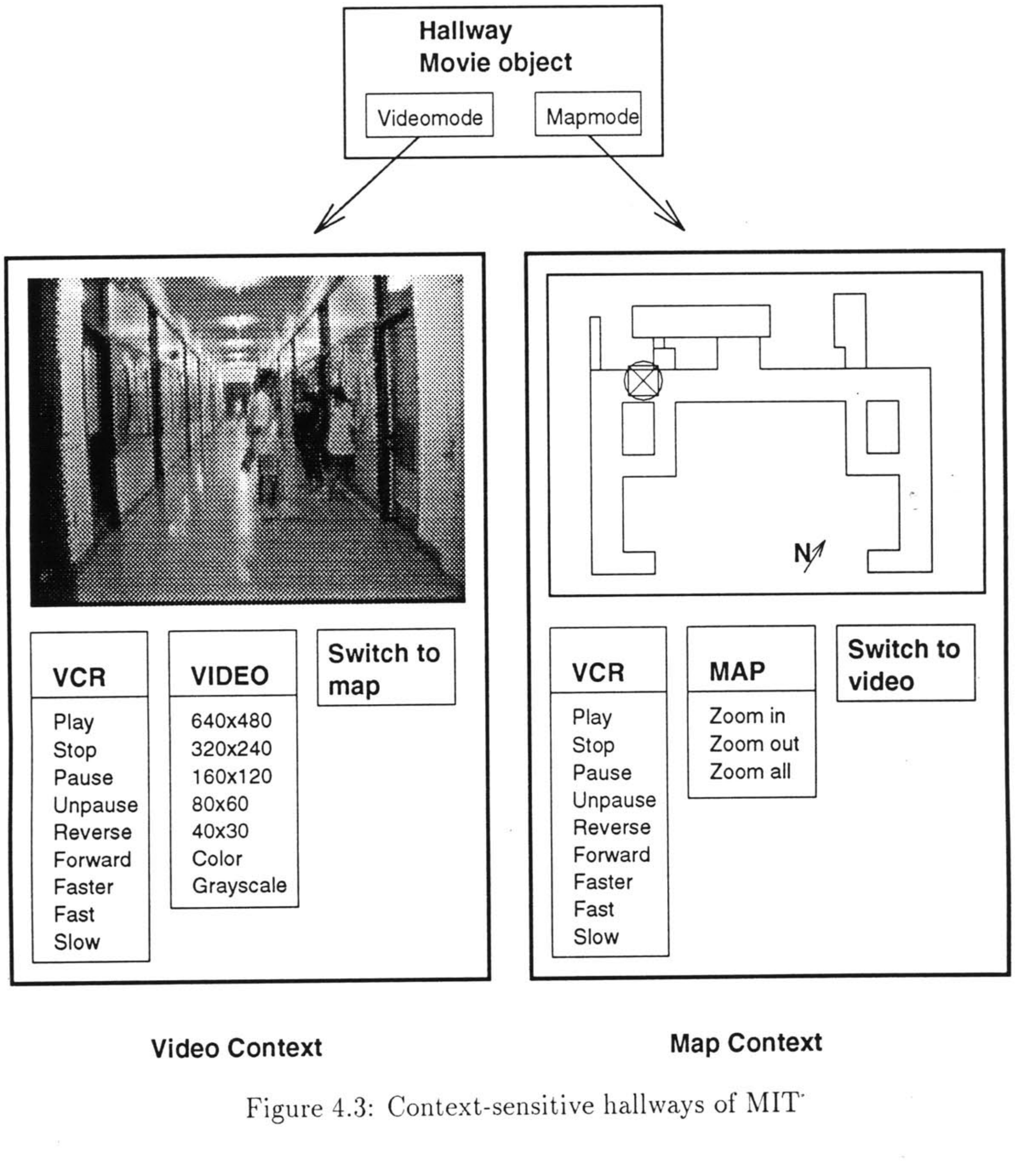

Figure 4.3: Context-sensitive hallways of MIT

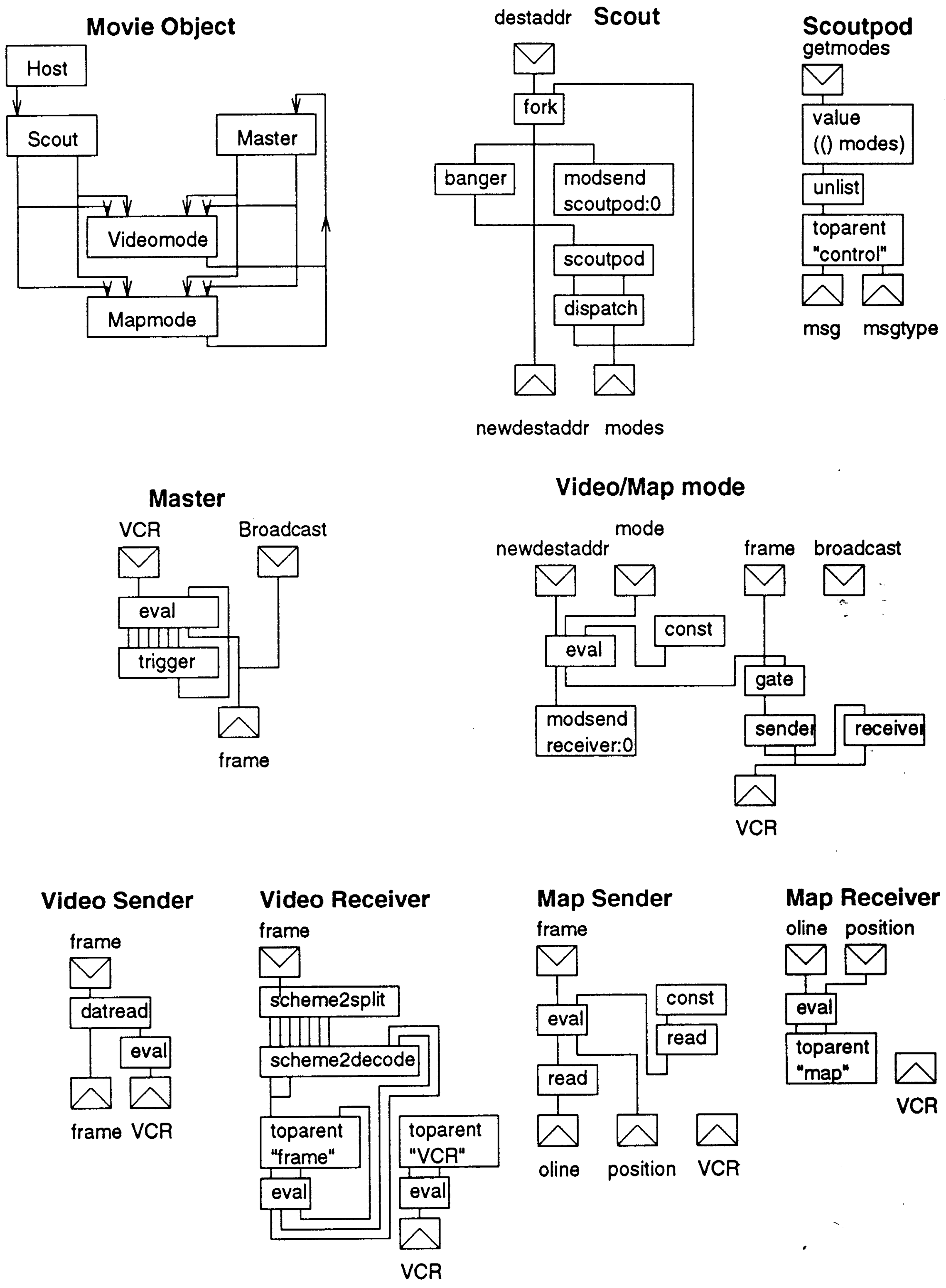

Figure 4.4: O schematic for the interactive hallways demo

There are three major improvements that can be made to the demonstration. The first is to be able to direct the movie object to both contexts simultaneously and synchronize the contexts, so that one can watch the video and follow along on the map at the same time. The second improvement is to use content cues to automatically adjust the video to the interests of the user. For example, if the user is a tourist then the video could highlight areas of tourist interest, but a physical plant worker would have the access panels and pipes highlighted for him. The third improvement is a matter of efficiency. Currently, the act of switching contexts requires several seconds (5-10 seconds on a DECstation 5000/120) and has not yet reached a "seamless" quality. Shortening this context-switching time would be a major improvement, although this may require considerable work on O itself.

Further (and hopefully, cleaner) demonstrations of context-sensitive movie objects will be created once the O widget for X has been written.

Chapter 5

Conclusion

Electronic multimedia has some major steps to overcome before it can establish itself as a natural means of communication. One of these steps is a consolidation of design methods into a rigorous field of study, a "core" of multimedia research. The current lack of such a "core" leads to a great deal of time lost to miscommunication and reimplementation of details. Once such a standard becomes widespread, designers can create, produce, and collaborate with greater efficiency. This may be the key to an explosion of new and exciting applications involving electronic multimedia, an explosion the computer world has been waiting for since its beginning.

Object-oriented programming is a method which has gained acceptance in the computer programming world. Designs can be realized much more intuitively, and work can easily be shared and reused, using object-oriented programming. Object-oriented programming is also scalable, and can be applied anywhere from the lowest level implementations to the highest level designs. However, object-oriented programming provides no less power than any other programming methods, but rather imposes a more rigorous standard for communicating that power.

Object-oriented design is clearly the next major step in computer multimedia production. Object-oriented programming, while not imposing any particular policy on look or feel or operation of multimedia, will still be valuable as a standard for how multimedia elements will interact with each other and their applications. If such a standard becomes accepted, then large libraries of multimedia objects can be built, shared, and distributed. Designers can create multimedia productions simply by fitting together the pieces they need, and creating the pieces they don't yet have.

Of course, this is all speculation and until object-oriented multimedia becomes a widespread method, it will be impossible to foresee all of the ramifications. In preparation, however, the Entertainment and Information Technology group has developed a system called O, which is a research tool for exploring the developments that are possible with object-oriented multimedia. O is only one of many possible object-oriented multimedia standards, and is not necessarily the best or most usable. In fact, O incorporates a couple of other research topics, distributed object processing and dynamic object location, and as a result it is quite large and unstreamlined.

Still, O has been able to shed some light on a couple of topics involving object-oriented multimedia. The first topic involves the representation of a multimedia object. Should that representation be text-based, or should it be graphically based? The answer turned out to be a combination of the two. The higher level designs work best if they are expressed as a process specification: functional modules connected together in a network, able to communicate with each other through messages. However, the operations of the functional blocks are best expressed in a more traditional text-based programming style.

Another topic that O has addressed is interaction between applications and multimedia objects. As the complexity of multimedia objects increased, O exposed a need for a standard policy of interfacing applications to multimedia objects. This led to the Host/OStream model which was based on the concept of isolation. Since multimedia objects are actually complicated running processes, they are separated from the application and given strict communication channels to the application, all to prevent an application from being endangered by a badly written object.

The Host/OStream model evolved further into the model for context-sensitive multimedia. This model proposed a new twist on the traditional model of applications and data. The context-sensitive multimedia model places most of the intelligence and the action within the multimedia objects themselves, while the application merely sets up an environment, or *context* for those objects to operate in. *Context-sensitive* refers to the object's ability to sense the conditions into which it has been placed, and adapt itself accordingly.

Further exploration of the context-sensitive model revealed some key points in how a context should describe itself to a context-sensitive object. The most important point was the separation between *contextual modes* and *content cues.* Contextual modes deal with selecting and configuring the actual elements of a multimedia presentation: video, audio, text, etc. Content cues allow the application to take a further step and specify preferences about the content of the presentation: what viewpoint to present from, what political slant to use, etc.

The final closure of the system was described by including a user-modeling system such as Doppelgänger. With such a system, the content cues of an application could be combined with the known traits of a user to form a presentation with a definite direction but tailored to the user's tastes. Furthermore, the user's interaction with the presentation could be fed back to the user-modeling system, allowing the user-modeler to update its model of the user.

Currently, context-sensitive multimedia is still in the design phase and very little actual testing has been done to see if its benefits are feasible. However, the idea has proven itself to be a sound extension of object-oriented multimedia. The question is, will context-sensitive multimedia in some form be an appropriate model for the creation of multimedia objects and multimedia applications? If not for all applications, which applications will benefit most from using context-sensitive multimedia?

Whether or not the methods presented in this thesis are entirely workable, the general idea for context-sensitive multimedia still remains sound. Context-sensitive multimedia advocates the creation of presentations that can adapt themselves to the system they are playing on and to the viewer who is watching them. This is an idea that will certainly grow more appropriate with the future, as not only multimedia systems diverge and evolve, but multimedia viewers also diverge and evolve in their uses and their expectations.

Chapter 6

Acknowledgments

The funding for this project was provided by **IBM**.

Thanks to the **Office of Facilities and Management Systems** for their assistance in obtaining floor plans of MIT in electronic form.

Thanks to the Media Lab's **System Services** for providing the use of AutoCad.

Thanks to the **Shadow Masks** softball team and the band **Adjective Noun**, for (unwittingly) providing video footage for use in the project.

Special honors to **Pascal Chesnais** for keeping up his support and manners through the duration of this thesis (and hopefully beyond).

Thanks to my advisor, **Walter Bender**, for his patience and his serendipitous but mighty insights into the work.

Thanks to my readers, **Tod Machover** and **Glorianna Davenport** for their time and their suggestions.

Thanks to my officemate **Judith Donath** and my office successor **Jon Orwant** for taking special care of my abstract.

Special thanks to my **family**, (yes, including my brother and sister), for their consistent phone calls, food, and interruptions that kept me going.

Finally, thanks to **M.I.T.** for the best college education and experience that I've had so far.

Bibliography

[Abr92a] Nathan S. Abramson. *The dsys Library*, April 1992.

[Abr92b] Nathan S. Abramson. *The Dtype Library*, June 1992.

[Abr92c] Nathan S. Abramson. *The O User's Manual*, July 1992.

[Ber90] Steven V. Bertsche. *HyperMedia/Time-based Document (HyTime) and Standard Music Description Language (SMDL) User Needs and Functional Specification.* University of Delaware, April 1990.

[BF92] Nathaniel Borenstein and Ned Freed. *MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies.* Bellcore and Innosoft, January 1992.

[Blo92] Alan Blount. *Bettyserver: More News Than You Can Beat With a Stick*, August 1992.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, 1990.

[Flu90] Fluent Machines Inc. *FM/1 Multimedia Development System*, 1990.

[IBM91] IBM. *AIX/Visualization Data Explorer/6000*, November 1991.

[Lau91] Brenda Laurel. *Computers as Theater.* Addison-Wesley Publishing Company, 1991.

[Lin92] Christopher J. Lindblad. *Thesis Proposal: A Distributed System for the Dynamic Manipulation of Temporally Sensitive Data.* PhD thesis, MIT Laboratory for Computer Science, 1992.

[NO90] Adrian Nye and Tim O'Reilly. *X Toolkit Intrinsics Programming Manual.* O'Reilly & Associates, Inc., 1990.

[Orw91] Jon Orwant. The Doppelgänger User Modeling System. In *Workshop on Agent Modelling for Intelligent Interaction.* International Joint Conference on Artificial Intelligence, 1991.

[Poo91] Lon Poole. Quicktime in motion. *Macworld*, September 1991.

[PZ] Miller Puckette and David Zicarelli. *MAX: An Interactive Graphic Programming Environment.* OPCODE Systems Inc.