Design, Specification, and Implementation of a Movie Server

by

Nathan S. Abramson

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Computer Science and Engineering

at the Massachusetts Institute of Technology

May 1990

Author_____
Department of Electrical Engineering and Computer Science
May 21, 1990

Certified by_____
Walter Bender
Thesis Supervisor

Accepted by_____
Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses

Design, Specification, and Implementation of a Movie Server

by

Nathan S. Abramson

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 1990

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering

Abstract

Modern digital video representations have strong ties to their ancestor representations in the analog film and video world, restricting digital video applications to those which can be expressed in a manner consistent with the analog world – frame based and time ordered. A new model for digital video is presented, which decouples representation from production, transmission, and display. Movies are treated as intelligent objects in an object-oriented system, giving the movies freedom to react or interact appropriately with viewers and the environment. A network movie server is designed, specified, and implemented as an illustrative application of intelligent and active movie objects. Movie objects are used to address the issues facing a movie server and demonstrate their power and flexibility in doing so.

Thesis Supervisor: Walter Bender
Title: Principal Research Scientist, Media Laboratory

This work was supported in part by the IBM Corporation

Acknowledgments

Pascal Chesnais is a knight in pink armor, a sage for confused minds and spirits, and a first class friend. With the gentleness of a cattle prod, he was there to help the birth of many good ideas and speed the burial of the rest. Without his continued presence, concern, and dedication, I would never have reached the level of ambition and growth that went into this document. For all this I am in deep gratitude.

Walter Bender was my advisor for this thesis, and I am grateful for his trust and confidence in my ability to work on my own and search for my own solutions.

I would like to thank the Garden, and the people in it, for its atmosphere and its undying hum, sometimes the only comfort in the strange hours of the morning.

I wish to thank James Tetazoo who drove me back to my thesis when other frivolities called, and called frivolously when the thesis began to dominate.

I thank my family for their undying support and prayers for me, which often comes in handy at a place like MIT.

Contents

1 Movies as Objects 1

1.1 Movies of the Present and Future 1

1.2 Problems with Modern Movies 2

1.3 Solution 2

1.4 Active Movies and Network Servers 3

1.5 Movies as Objects 4

1.6 Networked Movie Objects and Related Applications 4

1.7 Summary 5

2 Design of a Movie Application Server 6

2.1 Overview 6

2.2 Active Movies and Passive Data 8

2.3 Movie Application Model 9

2.4 Real-time Operation 11

2.5 Concepts 11

2.5.1 Active Movies and the Server Kernel 11

2.5.2 Description of Passive Movie Data 14

2.5.3 Communication Between Server and Client 15

2.5.4 Fluctuating Network Bandwidth 15

2.6 Designing Movies as Objects 16

2.6.1 Hierarchical Design of Movies 16

2.6.2 Using Standard Objects for Movie Design 17

2.6.3 Movie Objects and the Server Kernel 18
2.7 Operation 21

3 Specification of a Movie Application Server **22**
3.1 Elements of the Movie Application 22
3.1.1 Server 22
3.1.2 Client 23
3.2 Building the Movie Application 24
3.2.1 Source Files 24
3.2.2 Building the Movie Application from the Source Files 26
3.3 Movie Object Design 28
3.4 Data Descriptor Language 30
3.4.1 Overview 30
3.4.2 Data Types 30
3.4.3 Specifying Files 33
3.4.4 Labels 34
3.4.5 Constants 35
3.4.6 Data Descriptor Preprocessor 35
3.5 Object Oriented Descriptor Language 35
3.5.1 Object Oriented Model 36
3.5.2 Programming with the Object Oriented Extensions 36
3.5.3 Sending Messages to Objects 37
3.5.4 Creating objects 37
3.5.5 Defining a Class 38
3.5.6 Defining Methods 38
3.6 Communication Packet Extensions 39

4 Implementation of a Movie Application Server **43**
4.1 Data Description Language Preprocessor 43
4.2 Object-Oriented Extensions 45
4.3 Communication Packet Extensions 46

4.4 Network Interfacing 47

5 Future Considerations and Conclusion **48**

5.1 Future Considerations 48

5.1.1 Multiple Servers/Clients 48

5.1.2 Machine Independence 49

5.2 Conclusion 49

A Examples **51**

A.1 Sample Data Descriptor File 51

A.2 Sample Object-oriented Program 56

B Design and Implementation of a Reliable, Self-adjusting Network Protocol 60

List of Figures

2-1 Intelligence Coded into the Server 7
2-2 Intelligence Interpreted from the Movie 7
2-3 Intelligence Coded into the Movie 8
2-4 Active Movies and Passive Data 8
2-5 Indirect Access to Passive Data 10
2-6 Active Movies and the Server Kernel 13
2-7 Hierarchical Design of a Movie 17
2-8 "Splicing" with Linked Lists 18
2-9 "Circularizing" with Linked Lists 19
2-10 Data Stub Objects Accessing Data 20

3-1 Structure of the Server 23
3-2 Structure of the Client 24
3-3 Building the Movie Application 27
3-4 Structures of Data_Packets and Label_Sets 29
3-5 Example STRUCT Definition 31
3-6 Example ARRAY Definition 32
3-7 An ARRAY with VARIABLE size 33

Chapter 1

Movies as Objects

1.1 Movies of the Present and Future

Film and video dominate the media in modern society and will likely grow in the near future (through current technological trends). The majority of moving picture research is currently directed toward improving quality, widening distribution, and enhancing production. Other major directions of research aim to combine movies with computers in a personal computing environment.

Integrating movies into a computing environment will require that movies be stored, transmitted, and displayed in a digital representation. While analog methods exist for combining movies and computer output on a single screen, such techniques will quickly reach the end of their utility because of the computer's inability to directly manipulate analog signals. By using a digital movie representation, computers will have far more control over the manipulation and display of movies.

Numerous movie applications will become possible by making movies accessible to the digital domain. Transforming movies into a digital representation will open the world of computational video, in which all the power of computers can be applied to movies through networks, storage, display, analysis, and interactivity.

1.2 Problems with Modern Movies

Current representations of digital movies are based on the implementation of movies in the analog domain. In the analog domain, a movie is a time-ordered sequence of frames, stored on a passive medium (*e.g.* film or videotape) which is made active by an interpreter or projector. Current digital video representations adhere to this model by retaining the frame based representation and passive media storage (*e.g.* CD's, data files).

While this approach attains some short term goals such as improved distribution, storage, and manipulation, adhering to the analog movie model will preclude the creation of many new digital video applications. Modern frame based representations (DVI, MPEG) tightly couple production, distribution and display formats, limiting applications to those which can be expressed in a frame based manner from production to display. While this is not a problem for movie applications which are modeled after analog movies, there are other applications for which the frame is a confusing or inappropriate representation. Synthetic movies [10], in which sequences are constructed from many different movie and data sources, are difficult to model in a frame-based system since the output frames are combinations of frames from other sources. Model-based systems, which construct sequences from sources which are inherently non-frame based, will be difficult to express in a frame-based standard. Interactive movies [6] will also be difficult to design in a system where the movie representation is tied to a passive medium that is inherently noninteractive. Frames are also a difficult basis for current systems which require scalability and extensibility in space and time.

1.3 Solution

Rather than mapping the analog frame based standard onto digital video representations, a new and more flexible representation must be chosen for digital video data. The representation of the movie must be decoupled from the production, distribution, and display of the movie [9]. A representation which is more open to non-frame based systems and interactive systems must be incorporated. A model must be designed where digital movies may be described in complex and meaningful structures, as opposed to the simpler frame representation which physical film projectors require. Such a model should also allow digital movies to describe

their own *behaviors*, *i.e.* their reactions to changes in the viewer or distribution environment or to viewer commands. The new model should be able to encompass a wider range of digital video representations and will also be able to represent the intelligence necessary to govern the behavior of a movie.

A movie which contains some intelligence about its own behavior is no longer a passive entity like a file or a CD. The movie is an active element in its own presentation. The movie can take on the roles of both film and film projector, and do so more effectively than the old representation since *the movie knows its own content and can react intelligently to outside events.* This new representation for intelligence in digital video data is called an *active movie.*

1.4 Active Movies and Network Servers

The cost of storing and manipulating computational video is prohibitive for individual workstations, but can be offset by concentrating the required resources in shared servers. These servers can be connected to the workstations through a network, so movie applications will inherently be distributed systems. The servers are the central nodes of the distributed system, providing the resources for distributing movies to clients, and also serving as concentrators for broadcast video such as entertainment or news.

Distributed systems face many problems created by the chaotic nature of the underlying network implementation. The availability of bandwidth, a limited resource, may vary greatly as servers and clients load the system. A distributed movie application must accommodate these changes by attempting to maintain the quality of the display while remaining consistent with the intentions and content of the movie.

Active movies allow a distributed system to react appropriately to a chaotic network. The active movie model allows the movie to show itself, since the movie knows its own intentions and content best. Thus an active movie can be programmed to know exactly what to do in the case of a network load fluctuation. In the case of a passive movie, the server must bear the responsibility of interpreting the movie and trying to react in an appropriate manner, which is difficult for the server since the server knows little about the movie's content. The active movie offloads this responsibility from the server to the individual movie, which knows best how to handle a chaotic network environment for its own showing.

Thus the server acts as a resource manager kernel, which provides services to an active movie. Rather than acting as a projector and interpreter for a passive movie, the server takes the passive role and gives control to the active movie. While running, the active movie depends on the server to be an interface to the physical world of clients, networks, and filesystems.

1.5 Movies as Objects

An active movie can be expressed as an object in an object-oriented system, as described by [2]. Objects are pieces of data which know how to operate on themselves in response to messages. This concept fits the idea of the active movie, which contains structured digital video data and the knowledge of how to display that data while responding to viewer commands or a variable environment.

The object-oriented concept of hierarchical structuring applies well to recursively built movies, *e.g.*, scenes built of frames, sequences built of scenes, movies built of sequences. Message passing also works well with interactive movie applications. Viewer commands or notices about network load changes can generate messages sent to the movie objects which know how to respond to those messages.

1.6 Networked Movie Objects and Related Applications

The goal of integrating networked movie objects into the workstation environment is illustrated in part by several related works. While these applications do not specifically address the movies as objects concept, active movie objects will play a key part in bringing these works together into the movie playing workstation of the future.

Networked movie objects will help in building distributed synthetic movie applications. Synthetic movies act as a visual representation of the object-oriented paradigm perfectly suited to implementation using movie objects. This illustrates the ease of transforming an already object-oriented application into a movie application simply by adding movie sequences to the workings of the application.

Networked movie objects will provide a good opening for future distributed movie applications that diverge from the modern view of movies, such as model-based systems. Model

based movies are produced from sources which are based not on two dimensional frames, but on three dimensional models of a scene. Model based systems deal more concretely with the content of a movie, and less so with the movie's form. These systems will illustrate the flexibility of the movie object concept in building future applications.

1.7 Summary

Active movies are the representation needed to fully exploit the potential applications of computational video. Active network movie objects will provide the flexibility and power necessary to integrate distributed movie applications into the workstations of the future.

The remainder of this thesis will illustrate the advantages of active movies by describing the design and construction of an active movie server. This illustration will include both the design of an active movie representation, and speculations on the future development of the active movie model.

Chapter 2

Design of a Movie Application Server

2.1 Overview

A movie application is a distributed system with multiple connected elements, the servers and the clients. Like all distributed systems, movie applications are faced with the question of how to allocate the "intelligence" in the system. In the context of a movie application, the intelligence determines what sequences to show, when to show them, what to do if the network load changes, how to respond to user commands, *etc.*

Server machines will generally have more power and storage than client machines, which means that the intelligence in the system will be concentrated in the server rather than the client. It is possible that for some applications the client will actually be the more powerful machine. In such cases, the extra power in the client can be applied to such tasks as reconstruction, decoding, or analysis. However, the actual operations of the movie application will still be determined by the server.

One way to concentrate the intelligence in the server is to implement all the behaviors of the movie application in the server code, as shown in Figure 2-1. This approach suffers from inflexibility, since a new server will have to be written for each new movie application. This approach also suffers from a lack of clarity, since the behavior of the application is not clearly specified in one particular place but is instead buried within the code of the server.

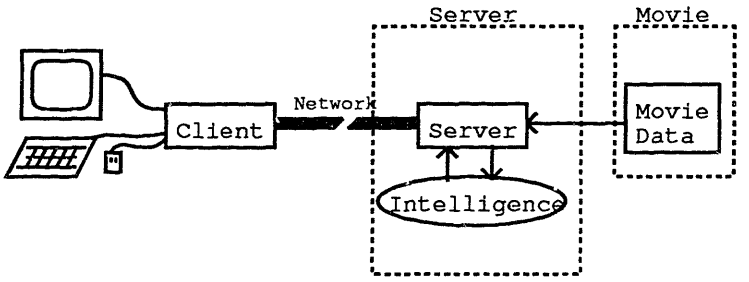

Figure 2-1: Intelligence Coded into the Server

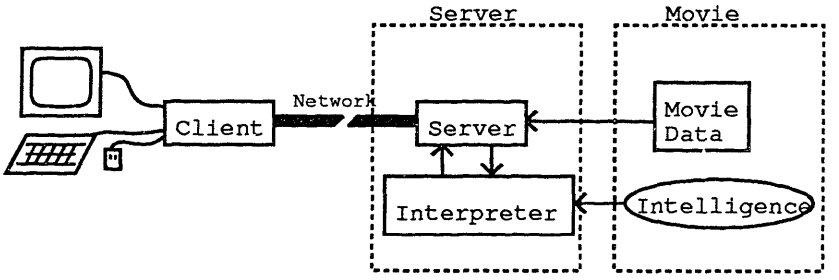

Figure 2-2: Intelligence Interpreted from the Movie

Another approach is to encode the behavior of the application into the movie data, and have the server act as an interpreter which translates the encoded behaviors into actual operations, as described by [5] (Figure 2-2). This approach is more flexible than the previous approach. However, this solution suffers from maintenance problems since new behaviors cannot be encoded into the data unless the server is updated to recognize those new encodings. This solution also suffers in performance, since a level of interpretation is required to turn encoded data into actual operations.

A better solution is to directly incorporate the behaviors of the movie application into the movie data, with no interpretation by the server (Figure 2-3). This solves the problem of maintenance since the server is only overseeing the execution of the movie and is not actually executing the movie itself. The extra level of interpretation is also eliminated, improving performance. With this approach, the intelligence in the system is not concentrated in the client, which simply forms the interface to the user, nor is it concentrated in the server, which

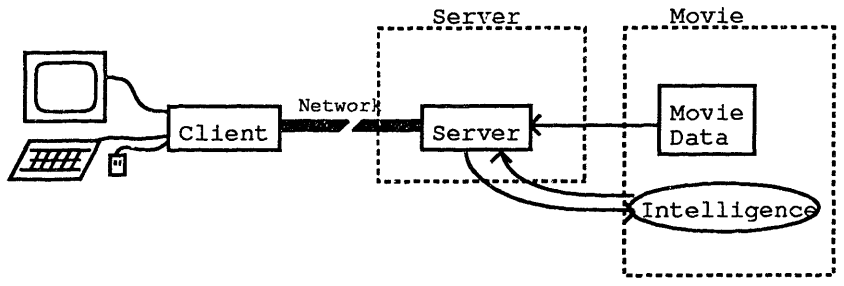

Figure 2-3: Intelligence Coded into the Movie

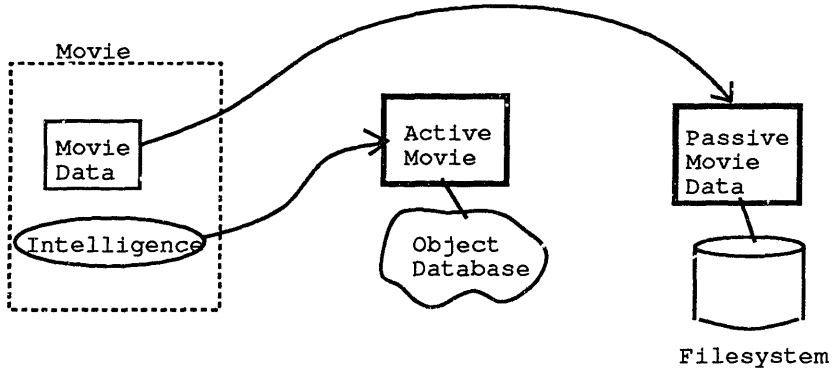

Figure 2-4: Active Movies and Passive Data

just acts as a resource manager and network interface. Instead, the intelligence is concentrated in the movie data itself.

2.2 Active Movies and Passive Data

Incorporating intelligence into the movie data divides the movie data into two parts: the "active" portion of the movie which contains the intelligence and behaviors of the movie, and the "passive" portion of the data which is the original or encoded movie data. These two portions are referred to as the *active movie* and the *passive movie data*, as shown in Figure 2-4.

The active movie and the passive movie data have different representations. The passive movie data retains whatever representation it had when produced by a digital video source

or coder, usually a series of structured files stored on the filesystem. The active movie is represented as an object database which runs as part of an object-oriented system. The server contains object-oriented support to interpret the object database representation of active movies.

When an active movie runs, it must have access to the passive movie data. Access is granted through a level of indirection which separates the active movie from the structure and location of the passive movie data. This removes the burden of calculating offsets and filenames from the active movie and allows the active movie to refer to data in more meaningful terms. For example, the active movie can say "Give me frame 3" without knowing where the frames are stored, what size the frame is, or what the frame's structure is. If the location, size or structure of the data changes, the active movie can still refer to that piece of data as "frame 3" without ever having to worry about the change.

To support this indirection, a translation mechanism stands between the active movie and the passive movie data. This translator knows the details of the passive movie data structures and locations, and thus can translate the conceptual references from the active movie into physical locations and offsets in the filesystem. Figure 2-5 illustrates the translator's role in making the structure and location of data transparent to the active movie.

The translator is informed of the structure and location of the passive movie data by a human-readable descriptor file. This concentrates *all* definitions of the data structure into a single source, providing the flexibility needed for a system where data formats or locations may often be changing.

2.3 Movie Application Model

An operating movie application consists of servers which handle the data and operation of the movie application, and clients which form the user interface of the application, displaying output and receiving input. The servers and clients may be arbitrarily connected as described by [3].

However, supporting such flexibility will complicate and obscure the basic design of movie applications. Thus, the design presented here will support a subset of this model, where a single client is connected to a single server. This one-to-one model simplifies the design of

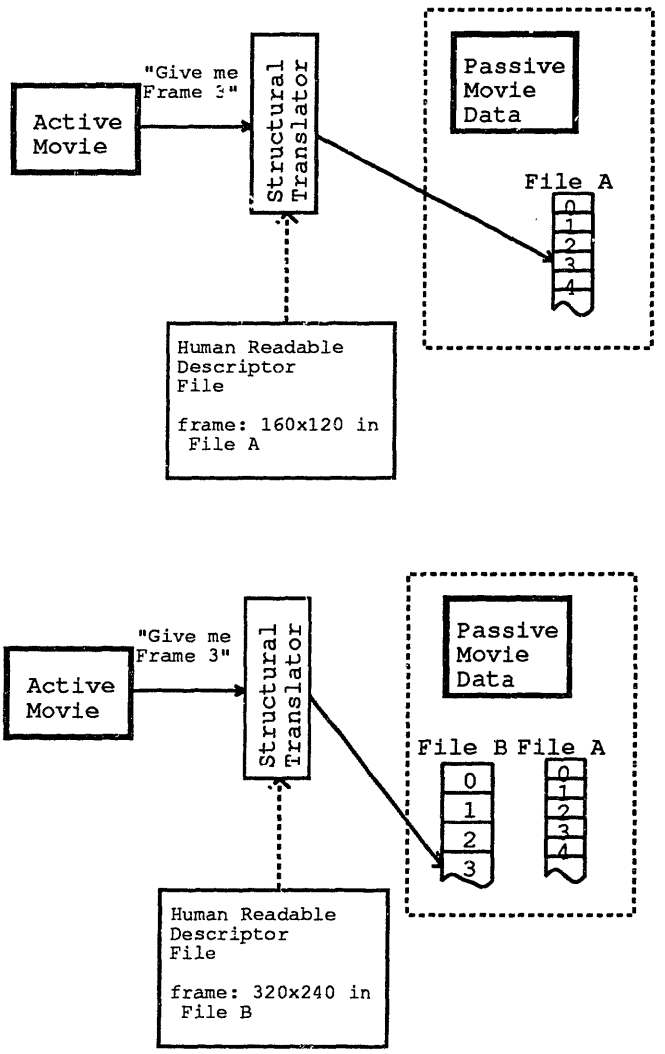

Figure 2-5: Indirect Access to Passive Data

certain features, such as protocols which require two way communication between server and client, or interactive applications which assume that a single user is operating the application.

The assumption is that changing a single server/client model to support multiple servers/clients will require only an expansion of the design presented here. However, this assumption may be untrue and multiple servers/clients might require an entirely new movie application design. This question is beyond the scope of this thesis, and remains to be answered elsewhere.

2.4 Real-time Operation

A movie application depends on synchronized real-time operation to support the narrow constraints of maintaining a particular frame rate [7, 8, 4]. However, a movie application cannot assume that its implementation will support real-time operation and synchronization, because the operating system and communication network are not necessarily built for real-time operation. While real-time operating systems and communication networks do exist, movie applications will probably be built for more common systems such as UNIX and Ethernet. Thus the movie application must be able to deal with a chaotic and non-real time implementation.

By communicating through a stream of many data packets, movie applications can account for a chaotic network and operating system. Even though the movie application cannot detect a timing problem until after the problem occurs, the application can correct the problem by modifying the remaining packets in the stream. This modification takes whatever form the application determines is appropriate, such as reducing the size of a few packets or eliminating some packets from the stream. Thus, even though exact synchronous real-time operation is not possible, real-time operation can be approximated by dynamically adjusting to unpredictable changes in the system.

2.5 Concepts

2.5.1 Active Movies and the Server Kernel

The movie server design centers around the design of the active movie. All the behaviors and operations of the movie applications are contained in the active movie. The active movie

communicates with the client to send movie data and receive viewer commands, and also communicates with the filesystem to access and manipulate passive movie data.

Active movies contain only operations which are relevant to the behavior of the movie application, such as how the application reacts to certain viewer commands, or how the application deals with increased network load. Issues which are dependent on the underlying implementation such as network communication and filesystem access are separated and grouped into the *movie server kernel.* This separation simplifies the design of active movies, allowing them to assume that the server kernel will be available to provide certain standard services. This design also improves the portability of active movies, since the definition of the active movie is separated from the implementation running the active movie.

While separating the active movie from the supporting implementation, the server kernel provides the active movie with an abstract model of the outside world with which the movie is communicating. The active movie no longer sees the network or the client, but works instead with virtual communication channels. The active movie also does not see the filesystem or files containing the data it needs, but accesses data blocks only through a naming system. The server kernel provides the services of translating virtual channels into actual network communication, and data block names into physical disk locations. As long as the active movie consistently uses the virtual channel and data block naming conventions, the active movie is insulated from the underlying implementation, and that implementation may be changed by changing only the server kernel and not the active movie.

Virtual Communication Channels

The server kernel provides the active movie with access to several virtual channels which represent communication with the client. The channels are unidirectional, running from server to client or from client to server. The channels from server to client are reliable and designed to carry large volumes of data such as video frames or lookup tables. The channels from client to server are unreliable and designed to carry smaller data packets such as commands.

The channels from server to client must be reliable to provide a reasonable abstraction to the active movie. If the active movie sends a piece of data such as a lookup table, the active movie must be able to assume that the data arrived at the client, otherwise following data such

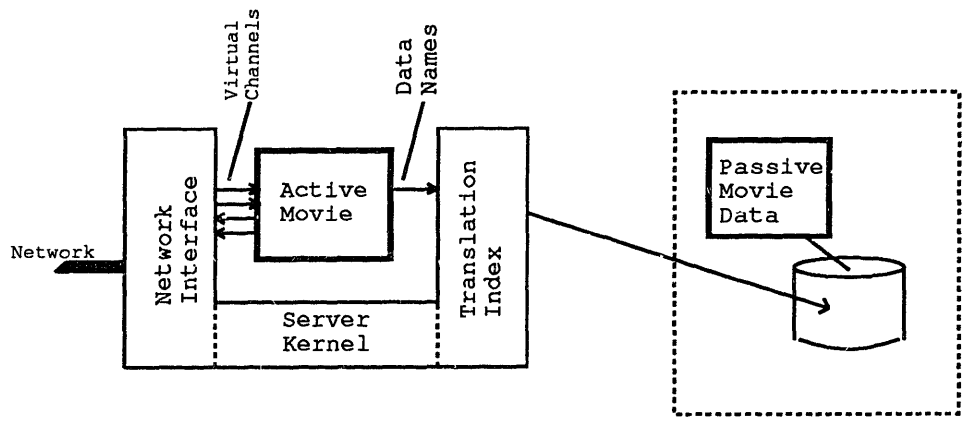

Figure 2-6: Active Movies and the Server Kernel

as frame data may be meaningless. Conversely, the channels from client to server need not be reliable. When the client sends a command to the server, there is no guarantee that the active movie will choose to service that command even if it is received, so the client must already be prepared to deal with command failures either through retries or error reporting. Placing a reliable connection from the client to the server will not change this situation. Since reliable connections are expensive to implement in terms of computational power (unless implemented in hardware), the desire for a reliable connection from client to server is not enough to justify the cost.

Reliability guarantees that data will be successfully transmitted, but the time required to transmit that data is unconstrained. This is a major problem with chaotic networks such as the Ethernet, especially for movie applications which inherently rely on real-time operation. As mentioned previously, the burden of approximating real-time operation is put on the movie application, not the communication system. The movie application must therefore be intelligent enough to correct for fluctuations in the operation of the network, even though the application is using a reliable communication channel.

Access to Passive Movie Data

The active movie accesses the passive movie data it needs through the server kernel, which uses a translation index to translate data names into filesystem locations. This separates the

active movie from the structure and form of the passive movie data, allowing that structure to change without changing the active movie. The active movie may also be configured to use different sets of data by changing only the translation index. This feature can be extended to use data sources other than a filesystem such as CD-ROMS or video cameras, all of which would be transparent to the active movie.

2.5.2 Description of Passive Movie Data

The translation index used by the server kernel is generated from a structural description of the files containing the data. This description lists the locations of the files containing passive movie data and defines the structures and formats of those files. The description is written in a data description language which is a human-readable method for specifying file formats. The data description can be created and edited using only a text editor.

A data description file must be generated for any set of data that is to be used in a movie application. To do this, the data must first be broken down into its basic organizational elements. For example, a DAT file is broken down into datasets, channels, dimensions, and primitive data types. Then, these organizational elements must be expressed using the data types offered by the data description language. These data types are very general, such as ARRAY and STRUCT, and so should be able to encompass any static data structure with uniform size fields.

Several data formats exist which do not have uniform size fields. These formats may be generated from sources such as variable bit rate coders, e.g. Huffman coders. The data description language also has provisions for describing these formats, by assuming that the data fields are tagged with headers which contain the size of the field.

Once a set of data has been specified in a data description file, labels may be added to the file. These labels will correspond to regions of data which can be accessed by an active movie. The active movie makes data requests by sending out data names. If a data name matches a label in the data descriptor file, then the active movie will be given the data region corresponding to that label.

2.5.3 Communication Between Server and Client

Communication links between the server and client are modeled as several asynchronous unidirectional virtual channels, running from server to client and from client to server. As previously explained, the channels from server to client are reliable, while the channels from client to server are unreliable.

Communication is carried in discrete packets, as opposed to stream communication which puts no natural boundaries around pieces of data. Packets which go from the server to the client are called data buffers, while packets going from the client to the server are called commands. The types and structures of packets are designed by the movie application and are known to both the server and the client. There are no "built-in" packet types or structures, although standard packet types may evolve with use.

While a movie application runs, packets are sent and received over the virtual channels. The type of a packet is not determined by a tag attached to the packet, but is instead determined by which virtual channel is carrying the packet. Thus only one packet type should be assigned to a virtual channel, and that assignment should remain constant throughout the movie application.

2.5.4 Fluctuating Network Bandwidth

Movie applications can be built on a shared network in which the usage of the network is chaotic and unpredictable, completely beyond the control of server or client. It is important that the movie application have some information about the network load at any point in time. With this information, the movie application can make a prediction as to how difficult it will be to get the next packet through, how much time it will take to get through, and whether the required frame rate can be maintained. If the network load rises to a problematic level where the application determines that packets will not go through in time, the application can respond in a way that the application determines is most appropriate.

Because the network is a dynamic system, measuring the instantaneous network load can be very difficult. Network performance is comprised of many factors, such as the number of packets per second, collisions, electrical interference, etc. The physical network does not have an easily defined set of state variables which can be put together to form the network load.

Even with the network load given, translating the network load into the predicted time for a packet transfer is a difficult and unreliable task. Thus, the network load is an unwieldy measurement for predicting packet transfer times.

A better measurement for predicting packet transfer times is a statistical combination of past packet transfer times. By observing the trends of the past few packet transfers, the future state of the network can be determined more accurately. This also has the advantage of an end-to-end measurement, in which all relevant factors such as network load, machine loads, device transfers, etc. are accounted for. A disadvantage of this scheme is that if the loads change suddenly and unpredictably, at least one packet will take longer to transmit while discovering this fact. However, once this condition is detected, the movie application can react quickly and adjust.

This approach requires frequent packet traffic in order to get a reasonable sampling of loads in the system. In a movie application, this will most likely be the case, since the application will probably be sending data as fast as possible. Also, if packet transfers are frequent, then sudden network changes will be less costly since the sacrifice of one packet will not be expensive if there are many packets being transferred.

2.6 Designing Movies as Objects

Because active movies are modeled as objects, object-oriented design techniques should be used for building movie applications. These techniques provide a standard basis for active movie design, because many active movie concepts can easily be expressed in an object-oriented manner.

2.6.1 Hierarchical Design of Movies

Hierarchical design is a fundamental tool of object-oriented programming. A hierarchy provides a clear organization of levels in a system and establishes standards for passing messages between levels. The levels also provide good abstract barriers for the system, making different levels invisible from each other except for messages passed through the levels. This allows an object which is sending a message to assume that the recipient will perform the desired

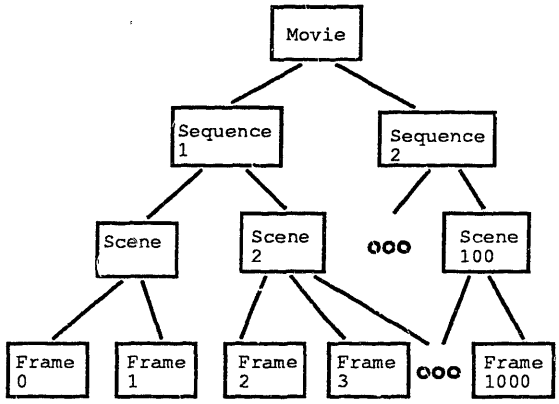

Figure 2-7: Hierarchical Design of a Movie

action, without the sender having to worry about how the action will be performed or knowing anything else about the recipient.

Most movie applications can be expressed in a hierarchical form. The primitive elements of the movie, usually the frames, will be placed at the bottom of the hierarchy. Each frame will be an independent object which knows how to display itself and how to adjust to a changing network load. Higher level objects such as scenes can be built out of the lower level frames. Scenes can be grouped into sequences and sequences grouped into a movie. Figure 2-7 illustrates this hierarchical design of a movie. This is only one example of how a movie can be organized hierarchically. If other organizations are appropriate, then it is likely that those organizations can also be specified hierarchically.

Object-oriented programming does not require hierarchical organization, and movies may be arranged in any arbitrary graph configuration if desired. In many cases, deviations from the hierarchy will be necessary, to at least account for global objects. However, these deviations should be kept to a minimum, since the hierarchical model does have several advantages as a clear, modular organization for movies.

2.6.2 Using Standard Objects for Movie Design

Object-oriented languages often provide standard object classes which represent useful abstractions. These abstractions can be applied to movie objects in building movie applications.

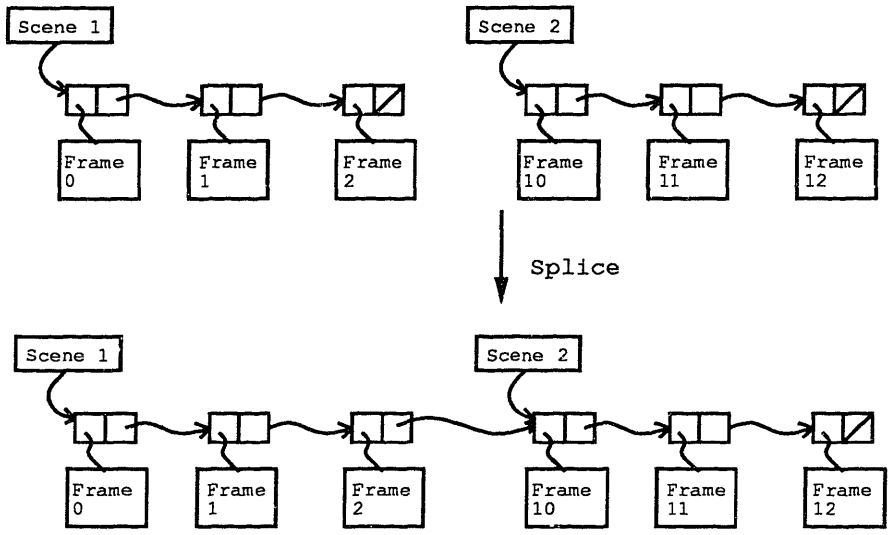

Figure 2-8: "Splicing" with Linked Lists

The linked list is a standard abstraction which movie applications will use. Linked lists are a good way to represent frame sequences. Operations on linked lists, such as splicing, joining, and "circularizing", can also apply to frame sequences, demonstrating the strong correspondence between frame sequences and linked lists.

Arrays, queues, and stacks are other typical standard object classes which can easily be added to the standard object libraries. When possible, movie applications should be built on these standard classes to provide a clear expression of concepts based on already existing abstractions.

2.6.3 Movie Objects and the Server Kernel

Movie objects must be able to communicate with the movie server kernel in order to access data from the filesystem or transfer packets to and from the client. Instead of using system calls, this communication can be handled in an object-oriented manner, by sending messages to objects.

Objects are used to represent the data which can be accessed on the filesystem. Each data region the active movie can access is named by a <label, dataset> pair. Every <label, dataset> pair available is associated with a "data stub" object, which is a standard object

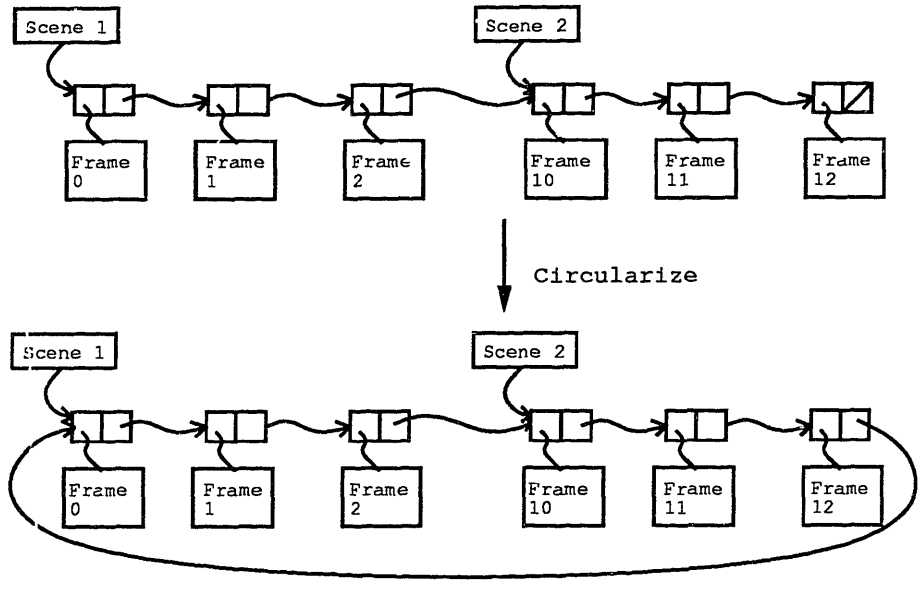

Figure 2-9: "Circularizing" with Linked Lists

class that acts as a handle for a data region. When a "data stub" object receives an "ACCESS" message, it will make a request to the server kernel to locate and read the data the object points to. The movie objects thus access the server kernel only through objects and messages. Figure 2-10 depicts data stubs accessing data from the passive movie data.

An object is also used to represent the client. Conceptually, the client is just an object which is operating remotely. Rather than going through system calls, the movie objects can treat the client just like any other object. Packets are sent to the client by sending messages to the client object, and packets are received from the client by polling the client object using query messages.

Objects are used as an interface to the other global services or information provided by the server kernel. This includes information such as network load or transmission statistics. By encapsulating these services into a single global object, the movie need only send messages to that object to receive services and information which would otherwise require system calls to access.

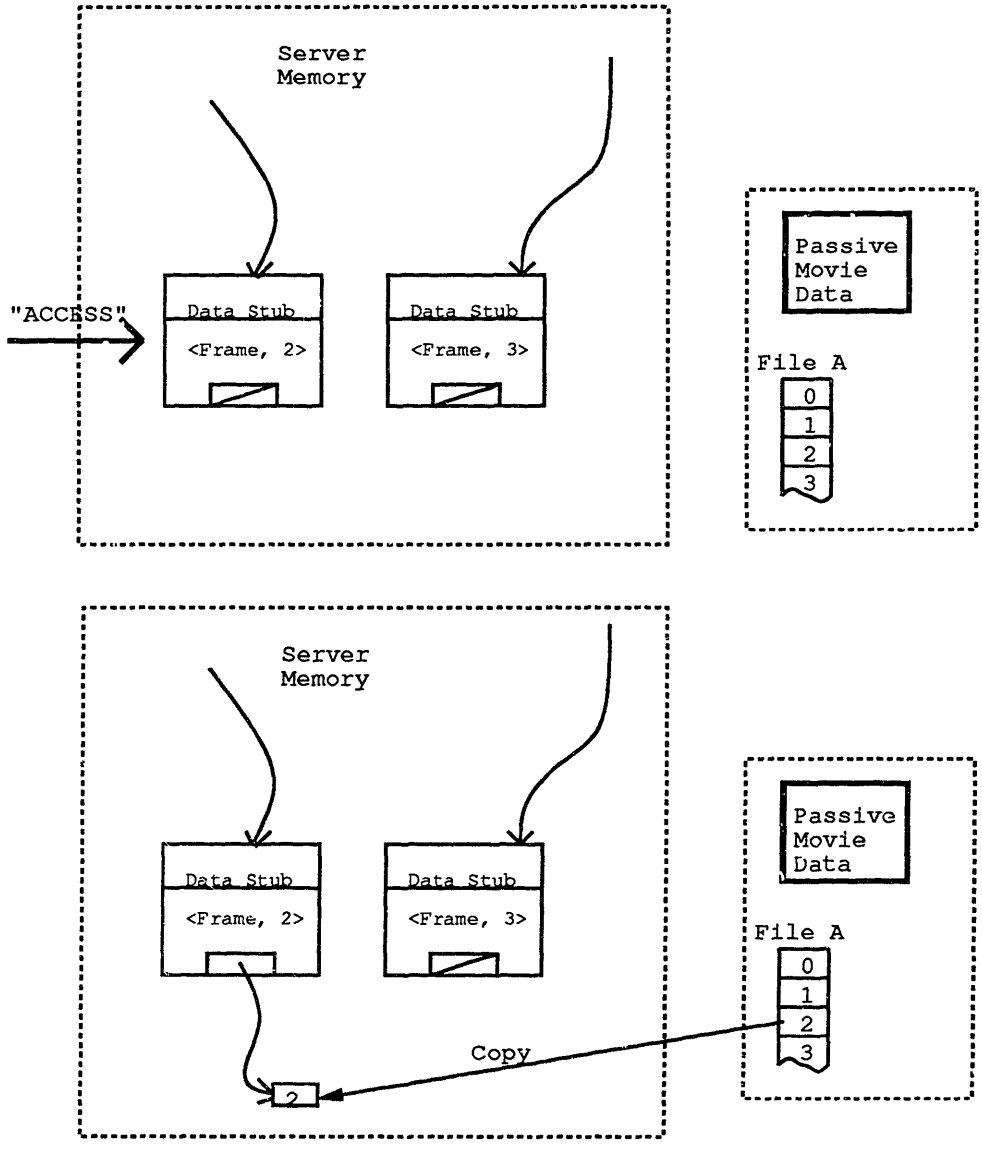

Figure 2-10: Data Stub Objects Accessing Data

2.7 Operation

Once the server kernel is built, an active movie application is designed, and the clients are implemented, the system is ready to begin operation. Initially, only the server kernel is running on the server machine. To begin a movie application, a client is activated and connected to the server kernel, which performs all initialization and starts the active movie running.

When a client connects to a server, the client specifies the set of active movie objects to be running as the active movie, and also specifies a data translation index to determine which set of passive movie data will be used. The server kernel receives the connection from the client, loads the specified data translation index, and starts running the objects in the specified active movie.

As the active movie runs, it generates requests for data and sending packets to the client. The server kernel intercepts the data requests and uses the loaded translation index to turn those requests into physical locations in the filesystem. The server kernel also intercepts the packets to the client and translates them into the proper protocols for transmission across the network. Commands sent by the client are intercepted by the server kernel, which stores the command packets until the active movie asks for them.

When the movie application decides to complete, it sends a termination message to the client. The connection between the client and server is broken, and the client terminates. The server kernel then removes the active movie objects and the translation index from memory, and waits for another connection from a client.

Chapter 3

Specification of a Movie Application Server

3.1 Elements of the Movie Application

A movie application consists of several subsystems, hardware and software. The major division is between the server and client subsystems, which operate on separate machines connected by an Ethernet network.

3.1.1 Server

The server consists of the *server kernel*, the *active movie*, and the *passive movie data* stored on the filesystem. The *server kernel* interfaces the active movie with the client and the filesystem. It contains a network interface which connects the virtual channels used by the active movie to the Ethernet network. The server kernel also contains a translation index which translates data name requests from the active movie into physical disk addresses on the filesystem. The *active movie* contains definitions for all the objects that will be used in the movie application. The active movie generates data name requests for the filesystem, sends data buffers to the client, and receives commands from the client. The *passive movie data* stored on the filesystem contains the raw or coded data generated by video sources and coding algorithms. The structure of this data is specified in the translation index which the server kernel uses to locate data requested by active movies.

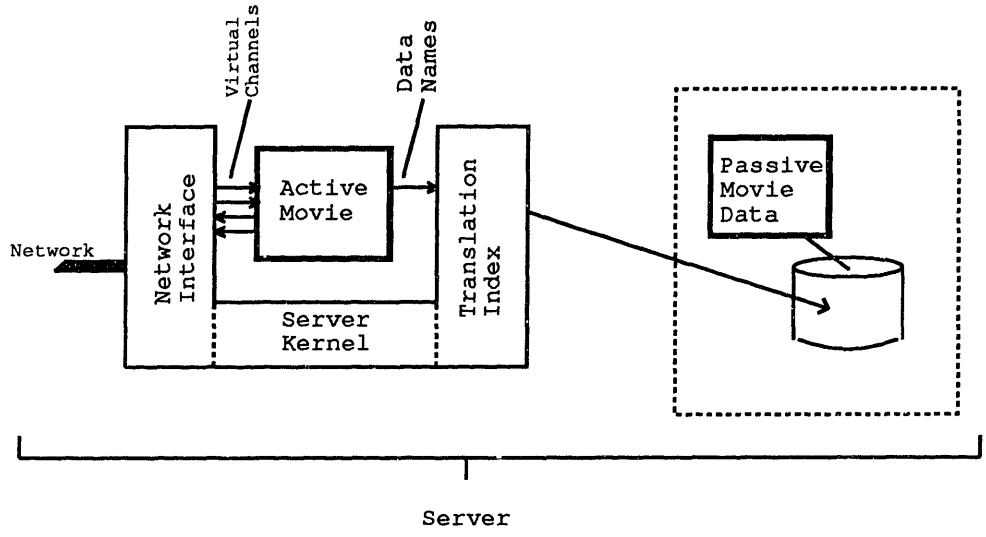

Figure 3-1: Structure of the Server

3.1.2 Client

The client consists of the *user interface devices*, the *network interface*, and the *client subsystem.* The *user interface devices* serve as input and output for the client subsystem. These include frame buffers, display devices, keyboards, and mice. The *network interface* handles all communication with the Ethernet, providing the client subsystem with access to the server through a set of virtual channels like those used in the server. The *client subsystem* contains the remainder of the client, such as the incoming and outgoing packet handlers, the user interface operations, and any reconstruction algorithms or hardware.

Some hardware implementations will combine reconstruction hardware with output devices, blurring the distinction between the client subsystem and the user interface devices. However, the basic client model still applies in these systems.

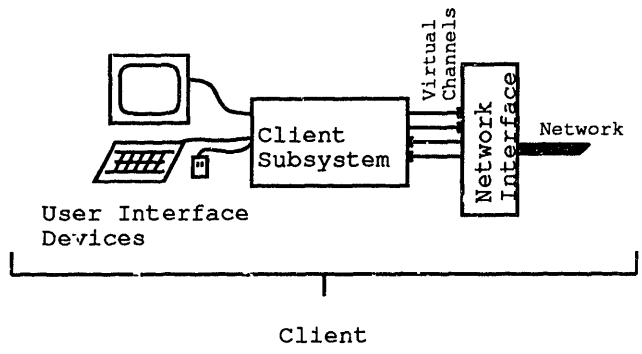

Figure 3-2: Structure of the Client

3.2 Building the Movie Application

3.2.1 Source Files

Four sources are required for building a movie application: the *active movie source file*, the *client source file*, the *data description file*, and the *data files*.

Active Movie Source File

The *active movie source file* defines the operations and communication protocols of the movie application. This file is programmed in C, with the object-oriented and communication packet extensions described later in this chapter. All object classes and communication packet structures are defined in this file.

The active movie source must contain a procedure called `run_movie()`. When the server kernel receives a connection from a client, the server kernel loads the requested translation index and calls `run_movie()`. `Run_movie()` creates and initializes all needed objects, then starts the objects running. When `run_movie()` completes and returns to the server kernel, the movie application terminates and the connection to the client is broken.

Client Source File

The client source file contains the program to be run on the client. This program is written in standard C and compiled for the machine which will run the client. A typical client will

follow this procedure:

1. Check for an incoming packet from the server – If a packet arrives, handle the packet by dispatching on the packet type. If the packet is a termination packet, exit the client.
2. Check for input from the user – If the user has entered a command, then send a command packet to the server
3. Repeat

The client knows the communication packet structures defined in the active movie source file. An include file produced from the active movie source file contains the structures of all command packets and data buffer packets, and also contains routines for transmitting command packets and receiving data buffer packets. This include file is used when compiling the client, so the client's communication definitions are consistent with the server.

Data Description File

All data that will be used by a movie application must be described in a data description file. Each data description file defines one entire set of data which can be used in the movie application. A movie application may use different sets of data by reading different data description files. Once a movie application has read a data description file, it will use that set of data until the movie application terminates.

The data description file is written in the data description language described later in this chapter. This language specifies the structure of data files in terms of simple recursively defined data types. Labels may also be inserted into the description to indicate the positions of data regions in the data files. When the active movie generates data name requests, those data names are matched against the labels in the data description file. If a match is found, then the position of the label in the data description file will correspond to the physical position of the data region on disk.

Data Files

The data files are the files which the active movie uses for its digital video information sources. These data files can contain information from digital video sources, data coded from digital

video sources, or any other data the movie application will require.

The data files must be stored in a format which is recognized by the data description language. For most formats this should not be a problem since the data description language encompasses many formats, especially those with uniform data field sizes. For data which contains variable bit rates or nonuniform data field sizes, the data description language contains specifications for how such data should be coded to be recognized by the data description language.

Different sets of data files may be used with the same movie application, by changing only the data description file. As long as the data description files contain the same labels, the active movie will not know the difference.

3.2.2 Building the Movie Application from the Source Files

A movie application is built by compiling the active movie source file and the client source file into the application code, and processing the data files and data description file into the movie data. Figure 3-3 outlines this process.

The active movie is the most complicated source file to process. This file is written in C, but also contains object-oriented material and communication packet definitions. An object-oriented preprocessor examines the active movie source file and translates all object-oriented and communication packet definitions into standard C code. This C code is then compiled with a standard C compiler and linked with the server kernel routines to form the server program. The preprocessor also produces a file called COMMPKT.H which is used in compiling the client.

The client source file is compiled with a standard C compiler on the machine which will run the client program. The client includes the file COMMPKT.H which was produced by the object-oriented preprocessor, which means that COMMPKT.H will have to be copied from the machine which processed the active movie source file to the machine which will compile the client source file.

The data description file for each set of data must be run through the data description preprocessor, which will produce a translation index for each data description file which the server kernel can use. When the preprocessor is run, it must have access to all the data

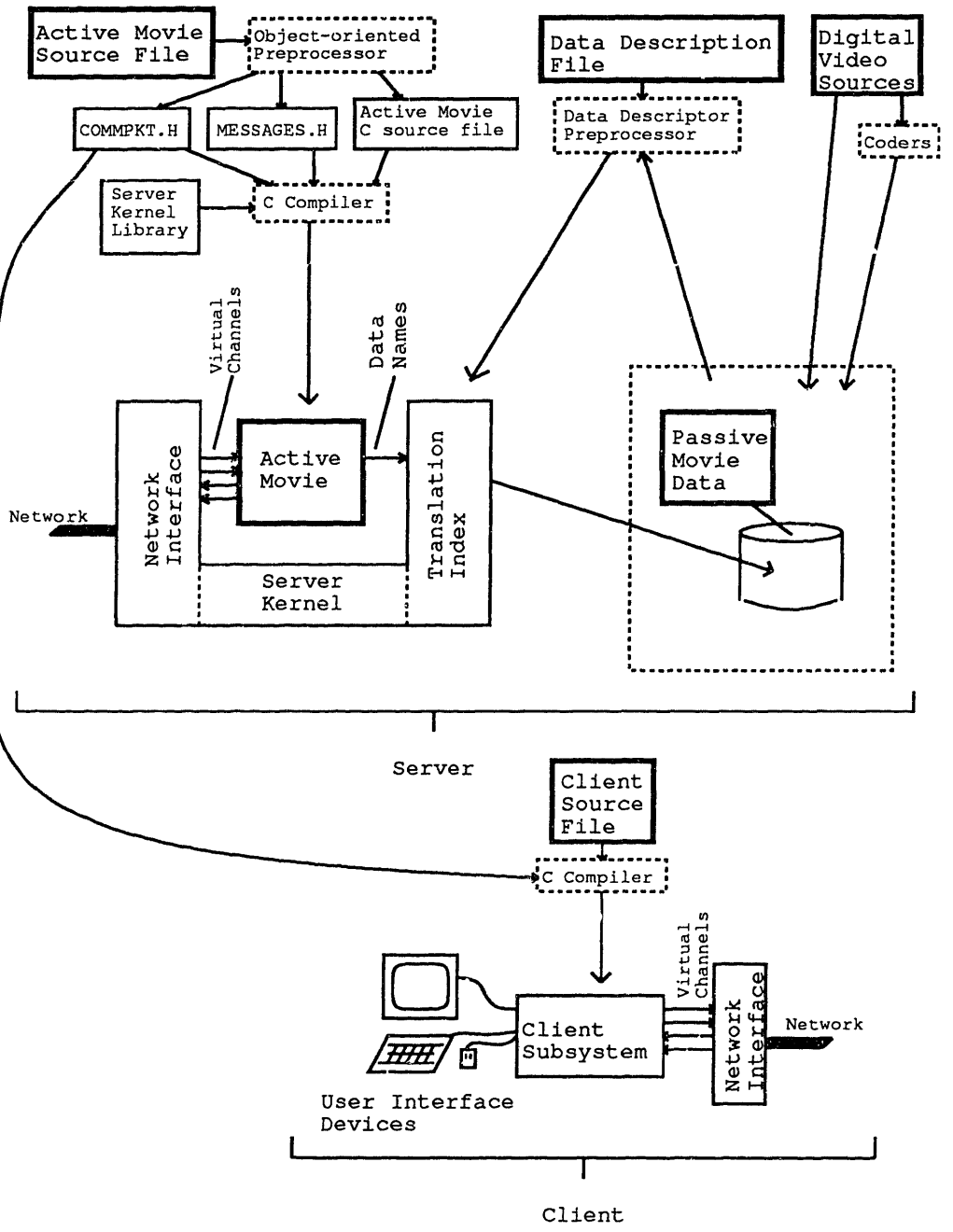

Figure 3-3: Building the Movie Application

described in the data description file to allow the preprocessor to resolve variable size data types.

3.3 Movie Object Design

There are several common approaches to designing a movie application as an object-oriented system. Predefined object classes are provided to take support these common designs. These predefined classes include arrays and lists. Another predefined class is the *data_packet*, which is used to access data from disk. A data_packet object is a "stub" for a data region, addressed by a *data name* (*i.e.*, a <label, dataset> pair). Data names are translated into physical disk regions by the server kernel using a translation index. Sending an "access" message to a data_packet object causes the object to load the data region corresponding to its data name and place the data in a buffer assigned to that object.

When the server kernel loads a data descriptor file, the server knows what <label, dataset> pairs are available for us , and creates a data_packet object for each <label, dataset> pair. The data_packet objects are grouped into a larger object class called label_set. A label_set is an array of data_packets, all of which have the same label. The label_set objects are then grouped into an array for all of the labels in the data descriptor file. Thus, all of the data_packets available to the application are grouped into one central object as shown in Figure 3-4.

Most movie applications will be frame-based, so a frame class of object is a good place to start for designing these applications. Each frame object should have the data and procedures it needs to reconstruct itself, and be able to respond to requests to degrade based on lowered bandwidth.

Not all frames need be the same. Some frames might degrade by dropping subbands, some frames might degrade by cropping the image, some frames might be color, some might be grayscale. The important thing, though, is that all different types of frames appear the same to the higher levels of the movie application. These higher levels should be able to send any frame object a "display" message and expect that frame to know enough to be able to send itself without the higher levels worrying about what kind of frame is involved. This is where using the object-oriented extensions comes into play. Under this system, each different kind of frame can be defined as a different class of object. However, all of the different classes

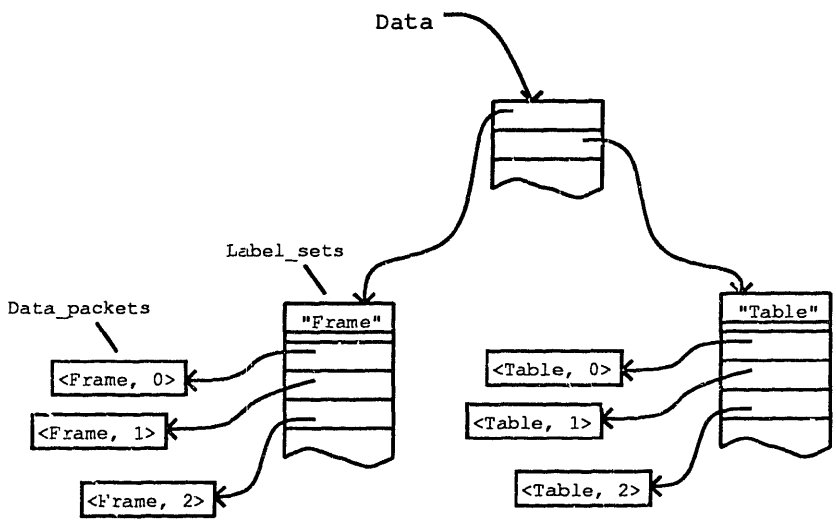

Figure 3-4: Structures of Data_Packets and Label_Sets

will be programmed to recognize the same "display" message, so to the higher levels all the different objects will look the same.

Once all the different frame classes have been defined and all the frame objects have been created, the frames can be grouped into higher level objects. For example, most movie applications involve linear sequences of frames, which are best expressed by the linked list abstraction. Lists allow great flexibility, since frames can be arranged in any order, and lists can be appended to each other or be made circular. Doubly-linked lists, which are predefined in the system, can also be traversed in both directions which makes lists perfect for showing a linear frame sequence.

Following this design method, it should be evident how to design higher abstraction layers such as scenes and sequences, all the way up to the top of the application which should be a movie object.

3.4 Data Descriptor Language

3.4.1 Overview

The purpose of the data descriptor language is specify the structure of files in a human-readable format, which can then be interpreted by the server kernel to translate data requests from the active movie into physical disk accesses. The data descriptor language specifies only the structure of data, without attributing any meaning to the data.

- The data descriptor model assumes that each file is made of a continuously stored series of datasets, where each dataset has the same structure.
- A dataset is made of data blocks which are stored continuously. Each data block is specified by a data type.
- The data type of a data block determines the structure and size of that data block, and also determines whether the data block is itself made of smaller data blocks (base or compound data types).
- Data blocks may be "tagged" with labels, which the server kernel will use to match against data requests from the active movie.
- Since a label represents a data block in all the datasets of a file, a <label, dataset> pair is necessary and sufficient for identifying one specific data block on the filesystem.

3.4.2 Data Types

The data descriptor language provides only three general data type primitives: One base data type called BASE, and two compound data types called ARRAY and STRUCT. Each data type represents a certain storage format, and is specified with a particular syntax.

BASE data type

The BASE data type represents a single n-byte block of data, where n is any integer greater than or equal to 0. A BASE type data block is primitive; it is not composed of smaller data blocks. The syntax for specifying a BASE data type is:

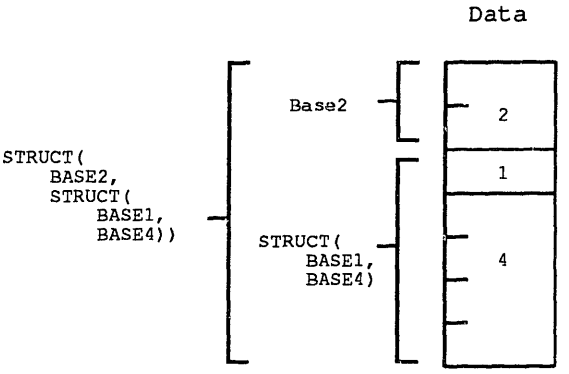

Figure 3-5: Example STRUCT Definition

BASEn

where n is the size of the base data type in bytes.

STRUCT data type

The STRUCT data type is a compound data type, meaning that it is composed of smaller data blocks which are stored continuously. The data blocks may all be of different data types. The data blocks are stored in the order specified by the STRUCT expression. The syntax for a STRUCT data type expression is:

STRUCT(<data type 1>, <data type 2>, ..., <data type n>)

where each <data type> is a BASE, STRUCT, or ARRAY data type. The total size of the STRUCT data type is the sum of the sizes of the data types contained by the STRUCT. Figure 3-5 shows an example of a STRUCT definition.

ARRAY data type

The ARRAY data type is a compound data type, composed of smaller data blocks which are stored continuously. Unlike the STRUCT data type, all of the data blocks in the ARRAY must be of the same data type. Each data block in the ARRAY is called an element. The syntax for an ARRAY data type expression is:

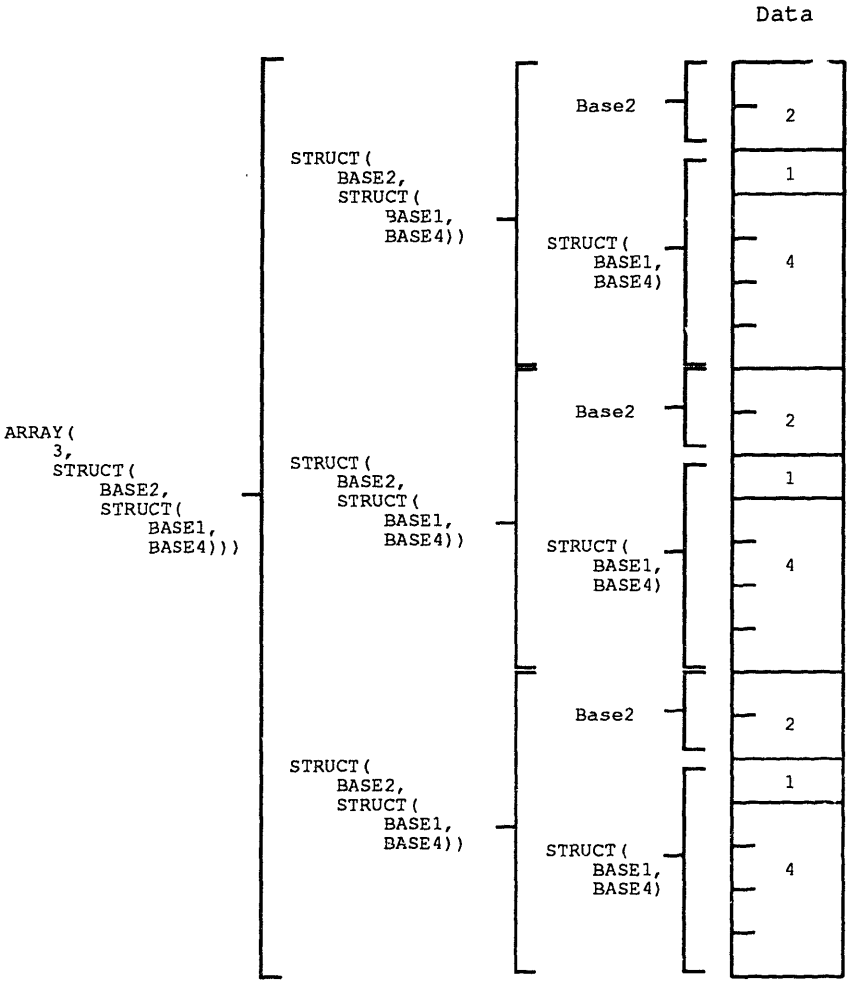

Figure 3-6: Example ARRAY Definition

ARRAY(<size>, <data type>)

where <size> is the number of elements in the ARRAY, and <data type> is the data type of each element in the ARRAY. The elements are stored and numbered continuously and in order, starting with element 0. The total size of the array is the sum of the sizes of each of the elements in the array. Figure 3-6 shows an example of an ARRAY definition.

Size may be an integer greater than or equal to 0, specifying exactly how many elements are in the array. There are two other possibilities for size, which are UNKNOWN and VARIABLE. Specifying UNKNOWN for size indicates that the number of elements is not known, but the

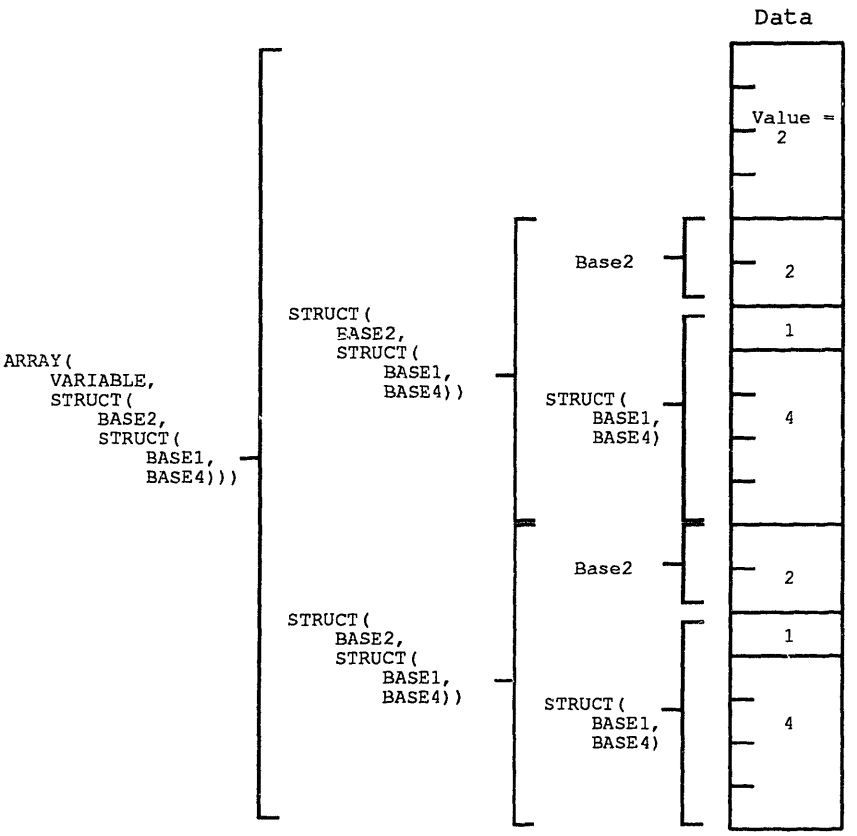

Figure 3-7: An ARRAY with VARIABLE size

array extends all the way to the end of the file, so the size of the file should be used to determine the size of the array. Specifying UNKNOWN for size can only be done in certain places, to be explained later. Specifying VARIABLE for the array size means that the size is not determined statically within the data description. Instead, the size is specified within the data itself. If the array size is VARIABLE, then the first four bytes of the array specify the number of elements in the array. These four bytes store a long integer, arranged in network order. Figure 3-7 shows an ARRAY with a VARIABLE size.

3.4.3 Specifying Files

The data specified by a data descriptor can span over several files. This is expressed in a manner consistent with the other data type expressions, using a special data type called

FILE. The syntax for a FILE expression is:

```
FILE( "filename", <data type> )
```

where "filename" is the pathname of a file in the filesystem, and <data type> is the type of data stored in that file. In the data descriptor model used, it is assumed that a file will contain several datasets each of which must have the same structure. This is expressed by requiring that the <data type> of a FILE be an ARRAY. The size of this array may either be statically determined (by an integer), or VARIABLE, meaning that the number of datasets is specified within the file, or UNKNOWN, meaning that the number of datasets is to be determined by reading datasets until the end of the file is reached. Thus, a file is specified by the expression:

```
FILE( "filename", ARRAY( <size>, <data type> ) )
```

where <size> is the number of datasets in the FILE, and <data type> is the type of each of the datasets.

A data descriptor will generally consist of several FILE expressions, listing all of the files which contain relevant data, the structure of those files, and labels to find data within those files.

3.4.4 Labels

Labels are pointers to data blocks within a dataset. Since data blocks are declared by data type expressions, a label can be attached to a data block by associating the label with the data type expression for that data block. The syntax for declaring a label is:

```
<label name>: <data type>
```

where <label name> is a string of up to 31 characters, containing upper/lower case letters, numbers, or the underscore character (_). <data type> is the data type expression (BASE, STRUCT, or ARRAY) for the data block to be associated with the label.

3.4.5 Constants

Integer *constants* may also be inserted into the data descriptor file. This is done by associating a label with a constant value. The syntax for a constant is:

```
<label name>: <value>
```

where <label name> is a valid label name as described above, and <value> is an integer value. In this declaration, the value of the label is an integer, not a pointer to a data block as with the other use of labels.

3.4.6 Data Descriptor Preprocessor

The data descriptor is stored in an ASCII file which is processed by a data descriptor preprocessor. This preprocessor interprets the ASCII file and determines the physical disk locations of the data blocks which have labels associated with them. The preprocessor also reads through the files specified in the data descriptor file to resolve locations and sizes determined by information in those files, *i.e.*, VARIABLE and UNKNOWN sizes for the ARRAY data type.

The preprocessor ignores all white spaces and C-style comments (enclosed by /* */).

The body of the data descriptor file contains constant declarations and FILE data type expressions. Labels are contained within the subexpressions of the FILE data type.

The end of the data descriptor file must be denoted by the END keyword.

Appendix A contains an example of a data description file.

3.5 Object Oriented Descriptor Language

Movie applications are written in C. To support the object-oriented movie model, an object-oriented environment has been extended to the C language. A preprocessor translates these object-oriented extensions into standard C code which can be compiled by a standard C compiler.

3.5.1 Object Oriented Model

The object oriented extensions are based on a simple model of object-oriented programming, using classes, methods, and message passing. An object is a set of data associated with a class. An object is referred to by a C pointer, meaning that references to an object may be passed around just like any other pointer. Objects are not operated on or modified by "outside" procedures. Instead, messages are passed to objects, and the objects themselves make modifications to their own data, return whatever data is requested, or perform side-effects such as I/O.

The class of an object determines two things: first, the class determines what data is stored by the object, and second, the class determines how an object will behave in response to passed messages. Each class has a "repertoire" of messages it recognizes and handles. The way those messages are handled are called the methods of the class. Each class has its own separate set of methods, which means that different classes can handle the same messages in different ways.

3.5.2 Programming with the Object Oriented Extensions

Since the object oriented extensions are just an addition to the C programming language, object-oriented programming may be mixed in with normal C programs. When the preprocessor goes through the file, the normal C material will be left intact, while the object-oriented material will be translated into standard C code.

Object Data Type

A special data type called "object" is used to represent objects. It is not necessary to know the actual structure of an object, but the object data type should be used to store objects and pass them around. Procedures which create objects return the object data type. Procedures used to pass messages take object types as arguments. Note that the object data type is used to represent all objects, regardless of their class.

Object declarations look just like normal C declarations. For example,

```
object myobj;
```

declares an object called myobj. When first declared, myobj has no value, just like any other C variable. Myobj can be given the value NULL, just like any C pointer:

```
myobj = NULL;
```

Or, myobj can be assigned to a new object by using an object creation procedure.

```
myobj = movie_create(...);
```

3.5.3 Sending Messages to Objects

Messages may be sent to objects using the `send_message()` command. The syntax for `send_message` is "`send_message(obj, message, arg1, arg2, ...)`", where `obj` is the object to receive the message, `message` is the name of the message, and `arg1, arg2, ...` are the arguments expected by that message (if any). When a message is sent to an object, the object looks to its class to see if a method is defined for that message. If so, then the method is executed on the object. A method may optionally return an object, which is the return value of `send_message()`.

3.5.4 Creating objects

Objects are created by the `<classname>_create()` command, where `<classname>` is the class of the object to be created. The return value of this procedure is an object. Thus,

```
myobj = movie_create();
```

will create a new object of the movie class and assign it to the object variable `myobj`.

When an object is created, an "initialize" message is automatically sent to the object. Any arguments included in the `<classname>_create()` command will be passed to the object when the initialize message is sent. Thus, every class must have a method defined for the initialize message.

For example, consider an account class, where an account object represents a bank account. The initialize method for accounts might expect a starting balance for its argument, so the command

```
myaccount = account_create(200);
```

would create a new object of the account class and assign it to myaccount, then send the equivalent of a "send_message(myaccount, initialize, 200);"

3.5.5 Defining a Class

A class is defined using a special syntax to indicate to the preprocessor that object-oriented material is being used. The expression "BEGIN_CLASS classname" will indicate to the preprocessor that a class definition follows. The BEGIN_CLASS expression must be the first expression on the line. After the BEGIN_CLASS expression, the data representation of the class is defined. This is done the same way a struct is defined in C, where each field of the data is given a name and type. After the representation is defined, the methods of the class are listed, each beginning with a "DEFINE_METHOD message" expression. After all the methods are listed, the class definition is terminated by a "END_CLASS" expression.

3.5.6 Defining Methods

A method is the set of instructions an object of a given class will execute upon receiving a particular message. Within a class, one method must be defined for each message that class is expected to recognize. A method is defined using the "DEFINE_METHOD message" expression, where message is a valid C name. This expression indicates to the preprocessor that the code following represents a method for the given message. The method will include all the code between the DEFINE_METHOD expression and either the next DEFINE_METHOD expression or the next END_CLASS expression. When an object receives a message, the system sees whether that object's class has a method defined for that message. If so, then the method for that message is executed.

A method has access to the data of the object which received the message. When a method is invoked, it automatically has a variable defined called "obj". This variable refers to the object which received the message, and can be used to access the fields of the data stored by that object. The data fields were defined at the beginning of the class definition, right after the BEGIN_CLASS expression. Obj is a pointer to a structure containing those fields, so the fields can be accessed using the usual "->" C operator.

A method may return a value, which must be of type `object`. This value is returned to the caller who sent the message to the object. A method may also return NULL if desired. The value is returned through the usual C `return()` statement.

When a message is passed to an object, arguments may also be included with the message. A method can tell the preprocessor what arguments it is expecting in its message through the "`METHOD_ARGUMENT`" command. The syntax for `METHOD_ARGUMENT` is "`METHOD_ARGUMENT(type, name)`", where `type` is a C type expression for the type of the argument expected, and `name` is the name of a C variable which will contain the value of the argument. The `METHOD_ARGUMENT` commands must appear directly after the `DEFINE_METHOD` expression, before any other statements in the method. The order of the `METHOD_ARGUMENT` commands must match the order of the arguments in the `send_message()` command which sent the message.

Appendix A contains an example of an object-oriented program.

3.6 Communication Packet Extensions

Creating new packet types requires a sizeable amount of code overhead: declaring the structure of the packet, writing code to transmit the new packet type, writing macros to access the fields of the packet, *etc.* The communication extensions allow the user to specify the structure of a packet type in C source code, and this specification will be translated into all the routines and macros necessary to support the new packet type.

There are two types of communication structures: *packets* and *commands*. Packets are sent from the server to the client, while commands are sent from the client to the server.

Packets contain a header followed by a data buffer. The header is made of fields, which are either integers or strings. Each field of the header is named. The header of a packet is always the same size and format, depending on what fields are in the packet. However, the data in a packet may vary in size, and the interpretation of that data is up to the client. Commands are similar to packets in that they contain a header. Commands, however, do not contain a data section.

A movie application may have several different types of packets and commands. The type of a packet or command is characterized by the type name, the fields in the header, and the channel number along which the packet or command will be transmitted.

To construct a new packet or command type, the following information is needed: the packet or command name, the channel assigned to that packet or command, the names of each field in the header, and the type of each field in the header (integer or string). For string fields, a size for the string must also be specified. For packets, it is assumed that the size of the data section will depend somehow on the data fields, so packets also need some sort of mathematical expression which will specify the size of the data section based on the header fields. For example, a frame packet might contain width and height as header fields, and the data size expression might be (width*height). Since commands have no data section, commands do not require a size expression.

To construct a packet or command, the following syntax is used:

for packets:

```
BEGIN_PACKET <packetname> <channel number> <size expression>
      <header field 1>
      <header field 2>
      .
      .
END_PACKET
```

where the header fields are of the form:

```
for integer fields:
      INT_FIELD <fieldname>

for string fields:
      STR_FIELD <fieldname> <string size>
```

for commands:

```
BEGIN_COMMAND <commandname> <channel number>
      <header field 1>
```

```
    <header field 2>
    .
    .
END_COMMAND
```

where the header fields are the same as the packets.

Once the packet or command structure is specified, the preprocessor will create several routines and macros for dealing with that communication structure. One routine created will transmit a packet or command. For packets, this routine is called "`send_<packetname>_buffer()`", and for commands the routine is called "`send_<commandname>_cmd()`". The arguments to these routines are values to be copied in to the packet or command header. One argument is expected for each field in the packet or command header, and the arguments are expected to be in the order that they were declared for the communication structure. In addition, packets require a final argument which provides the location of the data buffer. This argument is actually an object as defined in the object-oriented extensions. It is expected that this object accepts a message `get_data`, which takes an argument `char **buffer`, and that this message will set buffer to point to a data buffer containing the data to be sent.

The `send_..._buffer()` and `send_..._cmd()` routines will fill in a header with the data specified in the arguments. All integers are converted to network order. The header is then transmitted along the channel assigned to that packet or command type. For packets, the header is followed by the data buffer. These routines depend on a couple of global variables. The `send_..._buffer` routines require there to be a global variable named client of type `udpconnect`, which has been connected to the client. The `send_..._cmd` routines require there to be a global variable named server of type `int`, which is a descriptor for a UDP type socket connected to the server.

The preprocessor also creates several macros for accessing the fields of a packet or command. For packets, the macros are called `<packetname>_packet_<fieldname>`, and for commands the macros are called `<commandname>_cmd_<fieldname>`. For integer fields, these macros return the value of the field converted from network to host order. For string fields, these macros return a pointer to the string. Packets also have an extra macro called

<packetname>_packet_data, which returns a pointer to the data field of the packet. All of these macros assume that the packet or command has been put into a global buffer called comm_buffer.

Chapter 4

Implementation of a Movie Application Server

4.1 Data Description Language Preprocessor

The data description language preprocessor is built from a lexical analyzer coupled with a simple stack-based state parser, as described by [1]. The source code is written for the UNIX lex program, which translates the specified grammar into a C code lexical analyzer. The parser is also written into the source code.

The grammar for the data description language in Backus-Naur Form (BNF) is as follows:

```
<descriptor>: [<exp>]* END

<exp>: <comment> | <file_exp> | <constant_exp>

<integer>: [0-9]+
<comment>: /* [any character]* */

<type_exp>: <file_exp> | <array_exp> | <struct_exp> | <base_exp>
<type_exp2>: [<label> <type_exp>] | <type_exp>
<label>: [A-Za-z0-9_]+:
```

```
<base_exp>: BASE <integer>
<array_exp>: ARRAY ( <array_size> , <type_exp2> )
<array_size>: <integer> | VARIABLE
<dataset_exp>: ARRAY ( <dataset_size> , <type_exp2> )
<dataset_size>: <integer> | VARIABLE | UNKNOWN
<struct_exp>: STRUCT ( [<type_exp2> ,]* <type_exp2> )
<file_exp>: FILE ( <filename_exp> , <dataset_exp> )
<filename_exp>: " [any character except "]+ "

<constant_exp>: <label> : <integer>
```

The states of the parser are as follows, with format <token expected> (next state). The initial state is state 0.

```
0: FILE (1), label (26)
1: ( (2)
2: " [any character except "]+ " (3)
3: , (5)
4: ) (0)
5: ARRAY (6)
6: ( (7)
7: UNKNOWN | VARIABLE | <integer> (8)
8: , (10, push 9)
9: ) (4)
10: ARRAY (11), STRUCT (15), BASE (pop), label (25)
11: ( (12)
12: VARIABLE | <integer> (13)
13: , (10, push 14)
14: ) (pop)
15: ( (10, push 16)
16: , (10, push 16), ) (pop)
17-24: unused
```

```
25: ARRAY (11), STRUCT (15), BASE (pop)
26: <integer> (0)
```

As the parser interprets the data type expressions, it builds a tree structure of all the data regions from FILES to BASE types. Each node of the tree represents a data region, and each node has attributed to it a position and size. If the region is of a compound type (ARRAY or STRUCT), then the node will have children which represent the data regions making up the compound type. Each label encountered is put into a table with a pointer to the node it identifies.

After the parser has built the structural tree, the parser goes through and assigns a location and position to each node in the tree. For nodes which are of unknown size (VARIABLE size ARRAYs), the parser searches the data file where the ARRAY is stored and retrieves the ARRAY size from that file.

When all the nodes are filled in, the preprocessor generates an index file which matches the labels with the positions and sizes of the data regions they point to. The index file also contains the values of any constants defined in the data descriptor file.

4.2 Object-Oriented Extensions

The object-oriented preprocessor performs three basic operations: translating object class structure definitions into C structure definitions, translating object class methods into C routines, and assigning numbers to all defined messages.

Each object class method has a separate C routine generated for it, which performs some initialization then directly executes the code in the source file specified for the method. Also, each class automatically has generated an object creation routine and a dispatcher routine. The dispatcher routine receives an object and a message, and redirects the call to the appropriate method handler routine.

The structure of an object is a header followed by the object data. The header contains the size of the object, and also contains a pointer to the dispatcher routine assigned to that object's class. The header fields are automatically initialized by the object creation routine. When a message is sent to an object, the message is given to the dispatcher routine for the

object, which is identified in the object's header.

Each method is triggered by a message. As the preprocessor compiles the methods, it collects the messages and assigns a number to each one. If a message is encountered twice, only the first number assigned is used, and the message is not assigned a second number. When all the messages have been collected, they are written out as #define statements in a header file called MESSAGES.H.

4.3 Communication Packet Extensions

The same preprocessor which translates the object-oriented extensions also interprets the communication packet extensions. To create a communication packet, several steps are required. The structure of the packet must be designed as a C structure. The packet must be assigned to a virtual channel number. A routine must be created to transmit the packet. Various macros and pointers must be created to handle the packet. Some of this information, such as the packet structures and their channel numbers, must be compiled into the client.

The preprocessor takes care of all the above steps. As the preprocessor interprets a file, it produces a file called COMMPKT.H. When the preprocessor encounters a communication packet definition, all the necessary structures, routines, and macros are put into COMMPKT.H. COMMPKT.H is then compiled with both the server and the client code. C preprocessor switches are also included in COMMPKT.H so that only the appropriate portions of code are included in the server or the client.

A major problem in defining structures which will be transported between machines is byte ordering. Some machines will allow structures to be close packed, leaving no gaps between the fields. Other machines will require that the fields be aligned to 4-byte boundaries. The solution which was implemented was to make all fields have sizes which are multiples of 4. Long integers are used for the integer fields, and string sizes are rounded up to the next highest multiple of 4. Also, all the integer fields are placed before the string fields, although this may have no effect on the problem.

4.4 Network Interfacing

The network interface is designed to work with the Ethernet network, using the UDP protocol. The server/client model of connection is not used. Instead, when the server is idle it listens on an unconnected socket until a packet arrives. When a packet arrives, the server connects to the sender of the packet and assumes that the sender is the client.

The virtual channels are implemented by tagging each packet with a header containing the packet's size and its virtual channel number. This header is visible only to the network interface.

Chapter 5

Future Considerations and Conclusion

5.1 Future Considerations

5.1.1 Multiple Servers/Clients

The design presented in this thesis is built for a one server/one client model. This design will not be sufficient to represent multiple server/multiple client models. In some cases, adding multiple servers/clients will require a simple expansion of the design presented here. For other cases, this design will be completely unworkable.

Some multiple server/client applications will be collections of single server/client links operating synchronously. For these applications, the design in this thesis will require only minor modification for building many single server/client links together and synchronizing them.

However, some applications, such as broadcasting, will be difficult to model as a collection of single server/client links. For these applications, the presented design will be insufficient or inappropriate, and the most prudent route will be to redesign the system with the particular needs of the application in mind.

5.1.2 Machine Independence

Machine independent representation is an important issue which is not addressed in the present design. Different machines in the network will probably have different architectures and different low-level representations for data such as integers, floating point values, and structures. If two different machines attempt to communicate, there must be some translation mechanism which allows the machines to understand each other despite their differences.

Two issues are involved: detecting the need for translation, and performing the translation. To detect the need for translation, the machines involved must compare their architectures to see if they are different. More importantly, the architecture of the client must be compared against the architecture of the machine which created the data about to the served to the client. This exposes the need for tagging data with some indication of the architecture which produced the data. The current design would require little modification to account for this need.

The more open question concerns the actual translation between architectures. There are many possibilities for selecting who and when the translation will occur. The data might be preprocessed into a translated state, which would require more storage but would avoid the need for real-time translation. On the other hand, the data might be translated in real-time, by either the server or the client or both. The design presented in this thesis is open to all of these possibilities.

5.2 Conclusion

This thesis has suggested a new approach for representing computational video in a way which departs from the old frame-based standards imposed by the analog world. Digital movies in a computational environment will be capable of intelligent representations, representations which allow a movie to present and maintain itself without the need for a projector or interpreter. In light of currently developing applications and future ideas, digital movies must no longer be thought of as streams of interpreted data, but as objects which contain data and the ability to act on their own data.

Intelligent movie objects have demonstrated their power in the context of distributed

applications. The network movie server designed in this thesis illustrates the flexibility of using a movie representation which dynamically and intelligently reacts to a chaotic environment like a network. The server design also demonstrates that the movie object representation is an important step in the development of interactive digital video applications. Thus, intelligent movie objects have the power and flexibility to realize the full potential of computational video applications and are strong candidates for becoming the video representations of the future.

Appendix A

Examples

A.1 Sample Data Descriptor File

The following example illustrates all the features of the data descriptor language. The task is to describe a set of files with the following properties:

- file "/pictures/goodpicture/lut" contains 40 lookup tables. Each lookup table has a variable number of r,g,b triplets, where r,g,b are 8 bit values.
- file "/pictures/goodpicture/frames" contains 80 frames, together with "points of interest". Each frame is 160x120 1-byte indices. Each frame has one "point of interest", specified by an x,y pair of 2-byte integers. Each frame is preceded by the x,y pair for that frame.
- file "/pictures/goodpicture/captions" contains an unknown number of captions, where each caption is a variable size string of 8-bit characters. Each caption applies to a range of frames, specified by a starting frame and an ending frame, which are 2-byte integers. These values are stored following the string.
- There are three constant values: width, height, numframes. These specify the width and height of each frame, and the total number of frames.

The data descriptor file for this example could look like the following:

```
/* Example data descriptor file

   Describes three files, containing color lookup tables, frames,
   and captions.
*/

/* Constant declarations */
width: 160
height: 120
numframes: 80

/* Descriptor for file containing lookup tables.  Each lookup table
        is a variable size array of r,g,b triplets, where r,g,b are
        1 byte values. */

FILE("/pictures/goodpicture/lut",
      ARRAY(40,
            ARRAY(VARIABLE,
                  STRUCT(
                        BASE1,
                        BASE1,
                        BASE1))))

/* Descriptor for file containing frames.  Each frame is 160x120 1-byte
      index values.  Each frame is preceded by an x,y pair specifying
      a "point of interest".  x,y are 2-byte integers. */

FILE("/pictures/goodpicture/frames",
      ARRAY(80,
```

```
            STRUCT(
                  STRUCT(
                        BASE1,
                        BASE1),
                  ARRAY(120,
                        ARRAY(160,
                              BASE1)))))

/* Descriptor for file containing captions.  Each caption is a variable
      length string of 8-bit characters, followed by a starting frame,
      ending frame pair made of 2-byte integers. */

FILE("/pictures/goodpicture/captions",
      ARRAY(UNKNOWN,
            STRUCT(
                  ARRAY(VARIABLE,
                        BASE1),
                  STRUCT(
                        BASE1,
                        BASE1))))

END
```

Now that the structure of the files has been specified, labels can be added to allow access to the different data regions. Suppose the following labels are to be used:

lut points to the variable size color lookup table

frame points to the 160x120 frame

point_of_interest_x points to the x value of the point of interest

point_of_interest_y points to the y value of the point of interest

caption points to the variable length caption string

starting_frame points to the starting frame value of a caption

ending_frame points to the ending frame value of a caption

The data descriptor file would then look like:

```
/* Example data descriptor file

   Describes three files, containing color lookup tables, frames,
   and captions.
*/

/* Constant declarations */
width: 160
height: 120
numframes: 80

/* Descriptor for file containing lookup tables.  Each lookup table
      is a variable size array of r,g,b triplets, where r,g,b are
      1 byte values. */

FILE("/pictures/goodpicture/lut",
      ARRAY(40,
            lut: ARRAY(VARIABLE,
                  STRUCT(
                        BASE1,
                        BASE1,
                        BASE1))))

/* Descriptor for file containing frames.  Each frame is 160x120 1-byte
```

```
      index values.  Each frame is preceded by an x,y pair specifying
      a "point of interest".  x,y are 2-byte integers. */

FILE("/pictures/goodpicture/frames",
      ARRAY(80,
            STRUCT(
                  STRUCT(
                        point_of_interest_x: BASE1,
                        point_of_interest_y: BASE1),
                  frame: ARRAY(120,
                        ARRAY(160,
                              BASE1)))))

/* Descriptor for file containing captions.  Each caption is a variable
      length string of 8-bit characters, followed by a starting frame,
      ending frame pair made of 2-byte integers. */

FILE("/pictures/goodpicture/captions",
      ARRAY(UNKNOWN,
            STRUCT(
                  caption: ARRAY(VARIABLE,
                        BASE1),
                  STRUCT(
                        starting_frame: BASE1,
                        ending_frame: BASE1))))

END
```

A.2 Sample Object-oriented Program

This example will use the object oriented extensions to simulate a bank account. Bank account objects will be of class account. They will contain a balance, and an owner name. Accounts recognize the following messages:

initialize arguments int balance, char *name: sets balance of account to balance, and owner name to name.

print prints the owner and account balance

get_balance argument int *ret: Places the account balance in ret.

deposit argument int amount: Deposits amount dollars into the account

withdraw argument int amount: Withdraws amount dollars from account

transfer arguments object recipient, int amount: Withdraws amount dollars from the account and deposits it into recipient

split Creates a new account with the same owner. The money from the original account is split between the original and new account, and the new account is returned.

```
BEGIN_CLASS account
      int balance;
      char owner[33];

DEFINE_METHOD initialize
      METHOD_ARGUMENT(int, balance)
      METHOD_ARGUMENT(char *, owner)

      obj->balance = balance;
      strcpy(obj->owner, owner);
```

```
DEFINE_METHOD print
      printf("owner: %s  balance: %d\n",
            obj->owner,
            obj->balance);

DEFINE_METHOD get_balance
      METHOD_ARGUMENT(int *, ret)

      *ret = obj->balance;

DEFINE_METHOD deposit
      METHOD_ARGUMENT(int, amount)

      obj->balance += amount;

DEFINE_METHOD withdraw
      METHOD_ARGUMENT(int, amount)

      obj->balance -= amount;

DEFINE_METHOD transfer
      METHOD_ARGUMENT(object, recipient)
      METHOD_ARGUMENT(int, amount)

      send_message(recipient, deposit, amount);
      send_message(obj, withdraw, amount);

DEFINE_METHOD split
      object newaccount;
```

```
        newaccount = account_create(0, obj->owner);
        send_message(obj, transfer, newaccount, obj->balance / 2);
        return(newaccount);

END_CLASS


main()
{
        object bobsacct, jonsacct, jonsacct2;

        bobsacct = account_create(400, "Bob");
        jonsacct = account_create(250, "Jon");

        send_message(bobsacct, print);
        send_message(jonsacct, print);
        printf("Withdrawing $80 from Bob's account\n");
        send_message(bobsacct, withdraw, 80);
        send_message(bobsacct, print);
        send_message(jonsacct, print);
        printf("Depositing $70 to Jon's account\n");
        send_message(jonsacct, deposit, 70);
        send_message(bobsacct, print);
        send_message(jonsacct, print);
        printf("Transferring $30 from Bob's account to Jon's account\n");
        send_message(bobsacct, transfer, jonsacct, 30);
        send_message(bobsacct, print);
        send_message(jonsacct, print);
        printf("Splitting Jon's account\n");
        jonsacct2 = send_message(jonsacct, split);
```

```
    send_message(bobsacct, print);
    send_message(jonsacct, print);
    send_message(jonsacct2, print);
    printf("Withdrawing $60 from Jon's second account\n");
    send_message(jonsacct2, withdraw, 60);
    send_message(bobsacct, print);
    send_message(jonsacct, print);
    send_message(jonsacct2, print);
}
```

Appendix B

Design and Implementation of a Reliable, Self-adjusting Network Protocol

The goal for designing a reliable network protocol is to minimize overhead (extra packets on the network) when network transfers are occurring without error, and increase the overhead only when correcting for errors. TCP/IP, a standard reliable protocol, does not address this goal well. TCP/IP requires an acknowledgement for every fixed number of packets transmitted, thus assigning a fixed and expensive overhead even for perfect transfers. If errors occur, then the cost goes up even more to activate the retry protocols.

The main problem with TCP/IP is that the receiver is too constrained about receiving packets in order. The protocol makes sure that all the packets up to a certain point are received before moving on to accept new packets. Too many things can go wrong with this approach, e.g. a packet can easily get lost on the network, or the receiver may run out of buffer space or may still be processing a previous packet when a new one arrives. In these cases, the protocol demands that the lost packet be safely transmitted before accepting the next packet.

A better approach is to be more lenient about the order in which packets arrive, and deal with gaps only when absolutely necessary. By keeping a little state in the sender and receiver,

this is not very difficult to do.

For the Ethernet, the maximum packet size is around 1000 bytes, so large data buffers must be broken into packets of size 1000, and each packet must be transmitted separately. Ideally, the packets should all be sent as fast as possible, and the receiver will send a positive acknowledgement if the packets arrived safely. If some packets were missed, then the receiver should acknowledge by requesting those packets that were not received. This approach comes closer to fulfilling the goal of minimal overhead for perfect transfers, and minimal overhead increase for handling errors.

The solution implemented is based on these ideals. In this implementation, the server is continually serving packets and the receiver is processing them as fast as it can. When the receiver gets a chance, it sends back an acknowledgement message indicating what packets were received. If the receiver sees the same packet twice, then the receiver automatically sends the acknowledgement back. When the server receives an acknowledgement, it removes the acknowledged packets from its queue of packets to be sent. When all the packets are removed, then the server is finished serving its set of packets and may go on to serve the next set of packets.

There are some synchronization problems which can happen if packets are lost at critical times, such as when the receiver has received all the necessary packets but the server does not think the receiver has received them all because the final acknowledgement was lost. To add the needed synchronization, each group of packets is tagged with a buffer number. The protocol can then detect problems by noticing discrepancies between what buffer number the receiver thinks it is receiving and what buffer number is being sent. Once these problems are detected, the sender and receiver can correct for these problems in such a way that deadlock or livelock will not happen and communication can continue.

In this scheme, the correct timing is critical. Running everything as fast as possible will not yield optimal results, since the sender will probably be sending data far faster than the receiver can receive it and thus packets will be lost. There is, instead, a critical delay which must be inserted between the transmission of packets. If correctly set, this delay will allow just enough time for the receiver to process each packet before the next packet is sent, and the receiver will be able to fit in its acknowledgement at the correct time. This is theoretically

the fastest transmission possible over the network.

Determining exactly what the delay should be is an extremely difficult task since the conditions of the machines and network can fluctuate a great deal. Rather than predetermining the delay, the delay can be determined dynamically, which will in fact make the protocol self-adjusting to changing network conditions. The way this is done in the current implementation is to change the delay a little bit for every buffer sent and watch the effect. If the transfer rate increases, then keep changing the delay in that direction, otherwise start changing the delay in the other direction. The hope is that there is some stable point of maximum throughput around which the delay will settle. If the network conditions change, then the delay will settle to whatever the new optimal point is. Note that for this protocol to work well, there must be a large amount of data constantly being sent, so that the delay has a chance to "explore" and find its stable point.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] Steve Gano. Forms for electronic books. Master's thesis, Massachusetts Institute of Technology, 1983.

[3] Gunnar Karisson and Martin Vetterli. Packet video and its integration into the network architecture. *IEEE JSAC*, 1988.

[4] Gunnar Karisson and Martin Vetterli. Subband coding of video for packet networks. *Optical Engineering*, 1988.

[5] Andrew Lippman and William Butera. Coding image sequences for interactive retrieval. *Communications of the ACM*, 1989.

[6] Ralph Mayer. Personalized movies. Master's thesis, Massachusetts Institute of Technology, 1979.

[7] Robert Moorhead, Joong Ma, and Cesar Gonzales. Realtime video transmission over a fast packet-switched network. In *SPIE/SPSE Symposium on Electronic Imaging: Advanced Services and Systems*, 1989.

[8] Kazunori Shimamura, Yasuhito Hayashi, and Fumio Kishino. Variable-bit-rate coding capable of compensating for packet loss. In *Proceedings of SPIE: Visual Communications and Image Processing '88*, 1988.

[9] V. Michael Bove, Jr. and Andrew Lippman. Open architecture television receivers and extensible/Intercompatible digital video representations. *IEEE ISCAS*, 1990.

[10] John Watlington. Synthetic movies. Master's thesis, Massachusetts Institute of Technology, 1987.