An Implementation of a Simple DAVIC Server Using Orbix

Feng-Cheng Chang and Hsueh-Ming Hang

Dept. of Electronics Eng., National Chiao-Tung University
Hsinchu, Taiwan 30010, ROC

ABSTRACT

The purpose of this project is to develop a simplified DAVIC server on the Sun workstations under Unix and X window environment. DAVIC 1.0 is a comprehensive set of standards that define various types of end-to-end multi-media communication systems. More precisely, we implement only the high level server-client protocols and server service elements specified in DAVIC. This system can provide browsing service, download a file or a portion of it, and playback an MPEG sequence with VCR-like control. Limited by time, manpower and tools, not all the DAVIC specified elements are fully implemented. However, an implementation of a simple video server based on the DAVIC concept has been completed and demonstrated.

Keywords: DAVIC server, Video server, Orbix, Multimedia communications

1. INTRODUCTION

Multi-media services over digital networks becomes the latest trend in information communication. As the number of services grows, it becomes essential to have an unambiguous standard to develop components and systems. Many organizations and projects attempt to specify such a standard. DAVIC is one of these attempts and it seems to attract a lot of popularity recently. The DAVIC 1.0 Specification is released in December 1995. It is essentially a collection of existing standards plus some glue made up by the DAVIC group to bridge the gaps.

Our goal in this study is to implement a simple multi-media server that is close to the DAVIC specification. More precisely, our implementation focuses on the high layer protocols used in DAVIC. In the following sections, we describe what our system is and how it works.

2. DAVIC SYSTEM OVERVIEW

2.1. DAVIC System Reference Model

The *DAVIC System Reference Model (DSRM)* is shown in Figure 1. There are five entities in the DSRM: a *Service Provider system*, a *Content Provider System*, a *Service User System*, and two *Delivery Systems*. This figure also shows the information flows and related *reference points*.

2.2. The DAVIC Services and Server

The concept of a DAVIC server is simple. In the service domain, there are services which export interfaces to the client. The interface of a service is usually composed of several abstract interfaces. In a DAVIC server, there are several so-called *Service Elements* which provide interfaces to the clients.

Figure 2 shows the reference model of a DAVIC server. In addition to the network-related functions, the server has four *Core Service Elements*. There are three additional ancillary service elements which are not shown in the figure: Client Profile, File, and Download.

- **Service Gateway Element** The Service Gateway is usually the first element the client observes. It acts as a broker of all the services in the server. A client uses the interface of the Service Gateway to navigate the service domain and select the desired service.
- **Stream Service Element** The Stream Service Element is the repository and source for streams. Its interface provides the functions to control the media flow, reports the stream state, and optionally places events into the stream.

0-8194-2299-1/96/$6.00

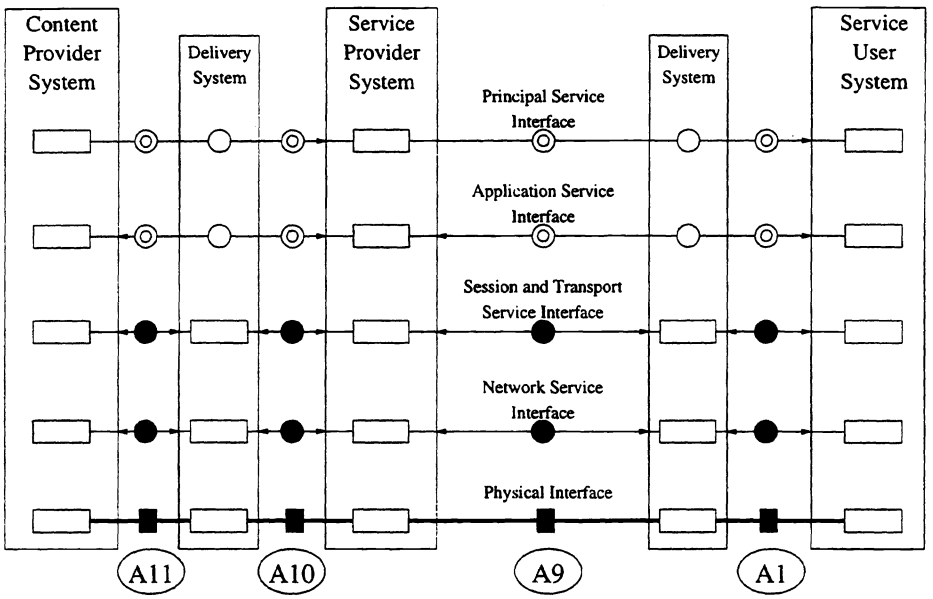

Figure 1: P1 Partition Level DAVIC System Reference Model[1]

- **Application Service Element** The Application Service is a general-purpose service that enables applications to operate via the definition, manipulation, and exchange of objects over their life-cycle.
- **Content Service Element** The Content Service is responsible for the interface to content management.

2.3. High Layer Protocols

The high layer protocol covers the service interface, the presentation protocol, and the session protocol.

Interfaces

The *Application Interface* is specific to each application while the *Generic Interface* is common to all the services. The Service Gateway provides the interface to explore the service domain. A Stream Service provides the interface to control media streams. A File Service interface provides file access functions. The Download Service, which is a part of Service Gateway, provides the interface to install software on the client device. To determine what software to install, the set-top device provides a profile of the available services.

Presentation and Session Protocol

DAVIC compliant systems require an interoperable protocol stack for the presentation and session layer that transports User-to-User primitives.

These protocols chosen by the DAVIC Specification are defined in the DSM-CC Specification and OMG UNO Specification for CORBA 2.0.

3. THE COMMON OBJECT REQUEST BROKER ARCHITECTURE*

3.1. The Object System

An *object system* is a collection of objects. In our context, objects provide the requesters (clients) one or more services. Each object has a well-defined interface which encapsulates its data representations and executable code. Hence, an object can isolate the implementation details from the requesters. The *interface* of an object is a description

*Material in this section is drawn mostly from.[2]

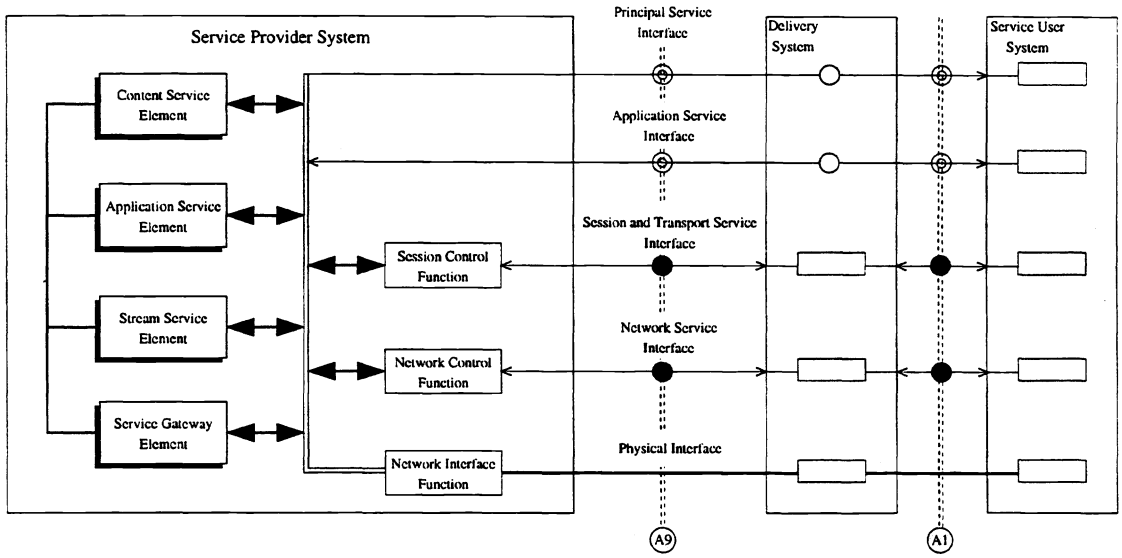

Figure 2: Server Reference Model[1]

of a set of operations that a client can ask the object to do. Interfaces is specified in *Interface Definition Language (IDL).*[2] A new interface can inherit from several existing interfaces and form a composed one.

In the *classical object model*, a client sends a message to an object, which then interprets the message to determine what service to perform. Thus there must be a method-selecting process before the object actually performs the desired operation. The selection of method can be performed by the mechanisms such as the *Object Request Broker (ORB)* described later. If something goes wrong during the handling of a request, an *exception* is returned to the client. An *operation* is an identifiable entity denoting a service that can be requested.

3.2. CORBA

The *Common Object Request Broker Architecture (CORBA)* is an *Object Request Broker (ORB)* technology defined by the Object Management Group (OMG). It provides mechanisms to make requests and receive responses between clients and servers. Using ORB, an application can communicate with different implementations of object systems on different types of machines in heterogeneous distributed environments. Based on the ORB technology, CORBA is structured to allow integration of a wide variety of object systems.

3.3. The Structure of ORB

Figure 3 shows the ORB structure. The interfaces are shown by striped boxes. The arrow indicates whether it is a call to ORB or a call from the interface. To make a request, the client either talks to the *Dynamic Invocation* interface, or talks to the *IDL Stub* interface. The client can also call the ORB interface for ORB-related functions. The Object Implementation receives a request from the *IDL Skeleton*. The Object Implementation may call the *Object Adapter* or ORB interfaces whenever needed. There is an Implementation Repository service which is specific to an ORB and contains information allowing the ORB to locate and activate implementation objects. The ORB Core is the part of ORB that provides the basic representation of objects and communication of requests. It uses the system-dependent elements to carry the representations.

3.4. The Integration of Foreign Object Systems

CORBA is designed to support different object mechanisms. When integrating a foreign object system into the ORB, three ways can be used. They are shown in Figure 4. When there exists an object adapter suitable for the

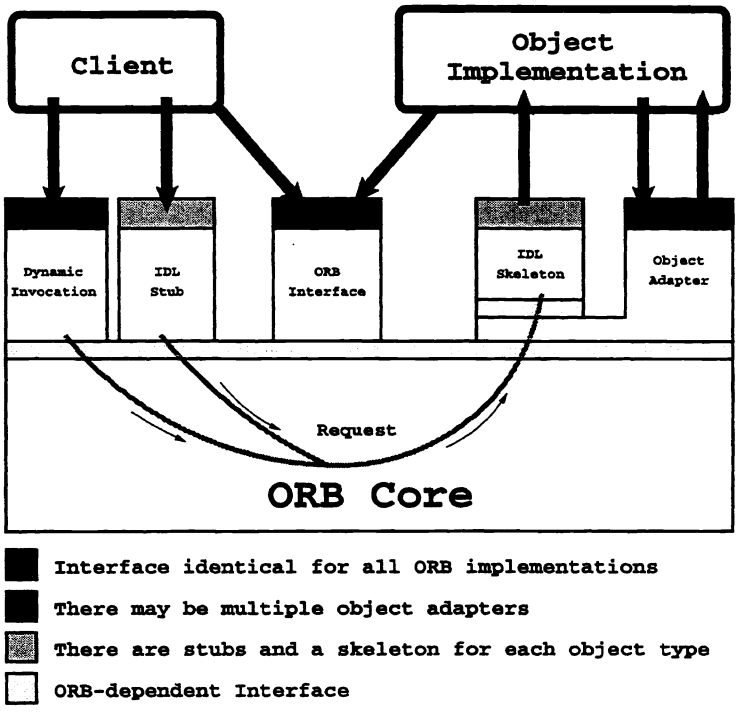

Figure 3: The Structure of ORB Interfaces[2]

object implementation, we simply hook the object to the adapter. If the object implementation requires special functionalities and there is no suitable adapter, we can design a new object adapter and use it to integrate the object implementation into the ORB. When integrating object systems with different object representations, we can use a gateway to alter the object representations between the two systems.

4. A SIMPLE DAVIC SERVER

4.1. System Overview

A fully DAVIC-compliant server is fairly complex. Under limited time and resources, we could only implement a portion of it. Before we jump into the details of programming, we first define what we mean a *simple DAVIC server.*

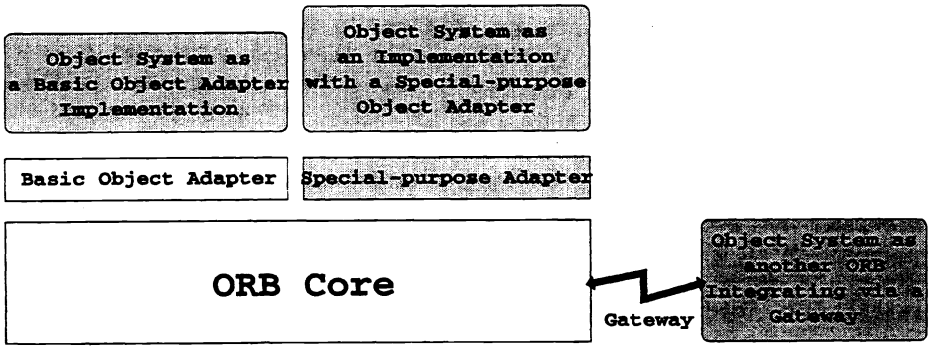

Figure 4: Different Ways to Integrate Foreign Object Systems[2]

In our laboratory, workstations are connected with Ethernet using the popular TCP/IP or UDP/IP protocol. It is convenient to build a simple DAVIC system on this well-constructed network environment. A client can connect to a server, browse up and down the service directories, choose the service it requests, and finally receive the service offered by the server. Here the word "client" means the end-user; hence, we only consider the relationship between the *Service Provider* and the *Service Consumer*. The intermediate *Delivery Systems* behave like the network routine functions to the Service Provider and Consumer. Since we focus on the S1 and S2 DAVIC data flow and let the existing computer operating system to handle communication problem, we do not need to implement a Delivery System. The following sections provide a more concrete viewpoint of the simple DAVIC system by means of interaction-diagrams.

4.2. Services in the Server

We have mentioned that there are four core services in a DAVIC server. The Content Service is used to manage contents. It is not possible for an ordinary client to use such a service. The Stream Service transports the media stream using non-Ethernet protocol, and it is difficult to implement this service in our laboratory environment. We thus design a new service to replace the Stream Service. Though it works differently from the Stream Service, it is easy to implement on Ethernet and it achieves the functions of a Stream Service by additional control mechanism on the user-side. The DAVIC specification adopts the DSM-CC definitions into its system. The Service Gateway related interface definitions can be found in the DSM-CC specification. In DSM-CC, a service called File Service is important. It provides the very basic file operations, and we implement this service as an application service. In summary, the services in our "simple DAVIC server" are:

Naming Context The service which allows the user to list, browse directories and resolve a new service.

BindingIterator When the directory list is too long, the user may get impatient waiting for the whole list to be transferred. A BindingIterator cooperates with a Naming Context to provide to the client a way to retrieve the trailing parts of a list.

Directory A special type of Naming Context, with additional functions to resolve multiple objects at once and get/put a whole directory tree from/to the server.

Service Gateway A special type of Directory service, with some network resource control functions.

File Read/write a segment of file starting from a specific offset.

MPEGFile This is the service to replace the Stream Service. It is a *file system based interface*, and it is a modification to the VDOBASE library which is described in section 5.4.1.

4.3. Functionalities of the Client

The client browser is used to interact with the server. One of the most important requirements of a browser is to present the service in a user-friendly way. The trend of user interface is graphical user interface (GUI). Most popular operating systems are companied with one or more windowing environments and libraries to help developers building GUI applications. In the simple DAVIC system we developed on our workstations, we build the presentation part of our client program using X window system. The client program is able to achieve the following targets:

- Connect to a specific server
- Display a service list on the screen
- Allow the user to select a specific service
- Update the service list automatically when needed
- Display a File Service GUI when invoking File Service
- Display a VCR-like control panel on the screen when invoking MPEGFile Service
- Display an image playback window on the screen when playing an MPEG video
- Allow multiple parallel tasking, e.g., we can down-load a file while playing an MPEG video

The client portion is developed to examine the operations of the server. Therefore, we use a simple and straightforward design based on the available tools (described in the next section). The client architecture does not follow the DAVIC specifications.

4.4. Interactions between Server and Client

We will use interaction diagrams to illustrate typical interactions between a server and a client. Figure 5 shows how a client connects to a server. The client uses the given host name to issue a connection to the specific server and obtains an object reference from the server. The client then uses the handle to request the top level service directory and display the service list on the screen. Figure 6 shows that the interactions to invoke a service. The server receives a request from the client. It tries to resolve a new object reference and returns it to the client. After the client receives the reference, it issues a request for a list and in the mean while it shows the newly created service on the screen.

5. IMPLEMENTATION

5.1. Tools

The implementation of our system consists two major parts of work. One is the CORBA mechanism, and the other is the representations of the interfaces on the client side. In this section, we will give a brief description of several programming tools involved in our implementation.

5.1.1. Orbix In order to build a DAVIC system, we need to implement CORBA. Orbix, produced by IONA Technologies, is a programming environment for building and integrating CORBA applications. One of its important features is that the components of these applications can run on different hardware nodes in a distributed system.[3] Orbix implements CORBA on Ethernet using TCP/IP and XDR. Since TCP/IP and XDR are ready-to-use protocols, Orbix implements ORB mechanism by providing libraries on both client and server side. There is a daemon process on a server node. It accepts all the incoming requests, and routes them to the desired object. If the object has not been activated, the daemon searches the implementation repository to find the registered service code of the interface and activates it.

When an object is constructed on the server side, Orbix creates a proxy (shadow) of the object on the client side. The client program operates the remote object through the proxy as illustrated in Figure 7. The proxy has exactly the same interface as the object at server, so the programmer can write the application as if the object was located in the local address space. This approach dramatically simplifies the distributed programming task comparing to the traditional RPC (Remote Procedure Call) approach. The traditional RPC approach is procedure-oriented, and it becomes harder to maintain the application when the service domain gets more and more complex. The concept of proxy makes the programmers free from worrying about where the objects are and thus reduces the efforts to port a normal program to a distributed one. When integrating objects on local and remote machines, the proxies enable programmers to write client codes with unified interfaces. However, if the programmers decide to use the traditional approach, they must distinguish between the conventional and the remote procedure calls. We will give an example of how to write codes on both sides in section 5.2.

5.1.2. Fresco Fresco is a joint development project under the auspices of the X consortium. Fresco supports *graphical embedding*, which binds structural graphics and application embedding together.[4] In order to achieve the goal of embedding, Fresco objects follow the CORBA standard object model, and as a consequence, integrating application objects distributed in a system is possible. Please refer to[4] for more detailed information about Fresco.

5.1.3. XForms The *XForms Forms Library for X*, or simply *Forms Library*, is a library of C-routines that allows a programmer to build up interaction forms (windows) with buttons, sliders, input fields, dials, etc. in a very simple way.[5]

5.2. File Service

In this section, we will describe the implementing details and coding styles of using Orbix.

5.2.1. Using Orbix Before implementing the applications, we need to use the Orbix IDL compiler to generate a set of C++ files. Then we write an implementation C++ class of File; here, we call it "File_i". Orbix provides two approaches of writing implementation classes:

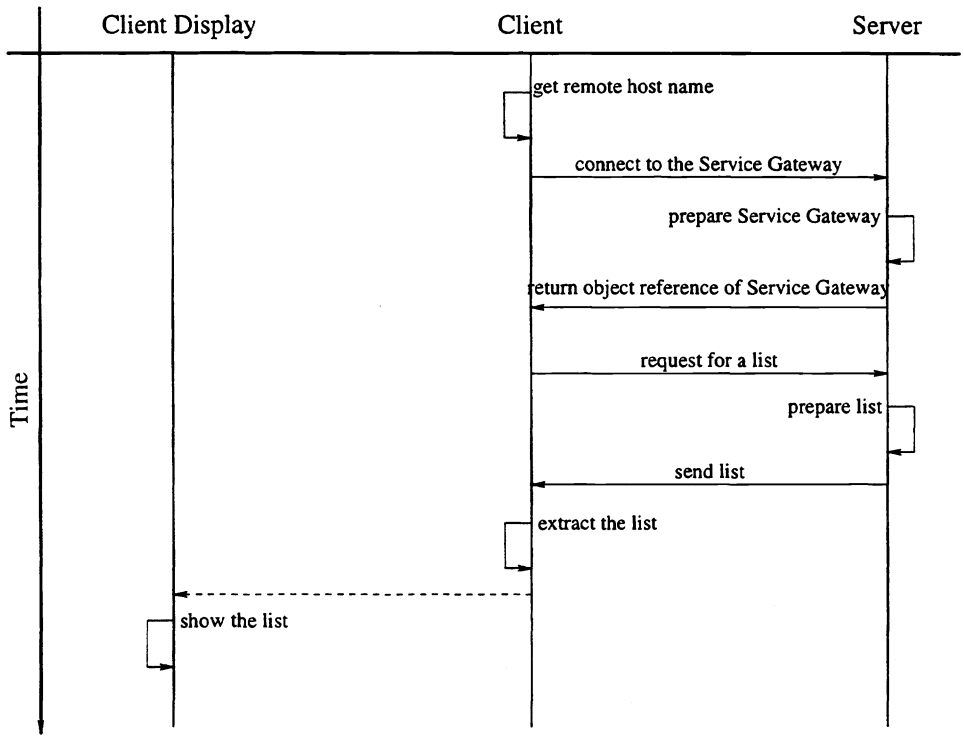

Figure 5: Interactions of Connecting to a Server

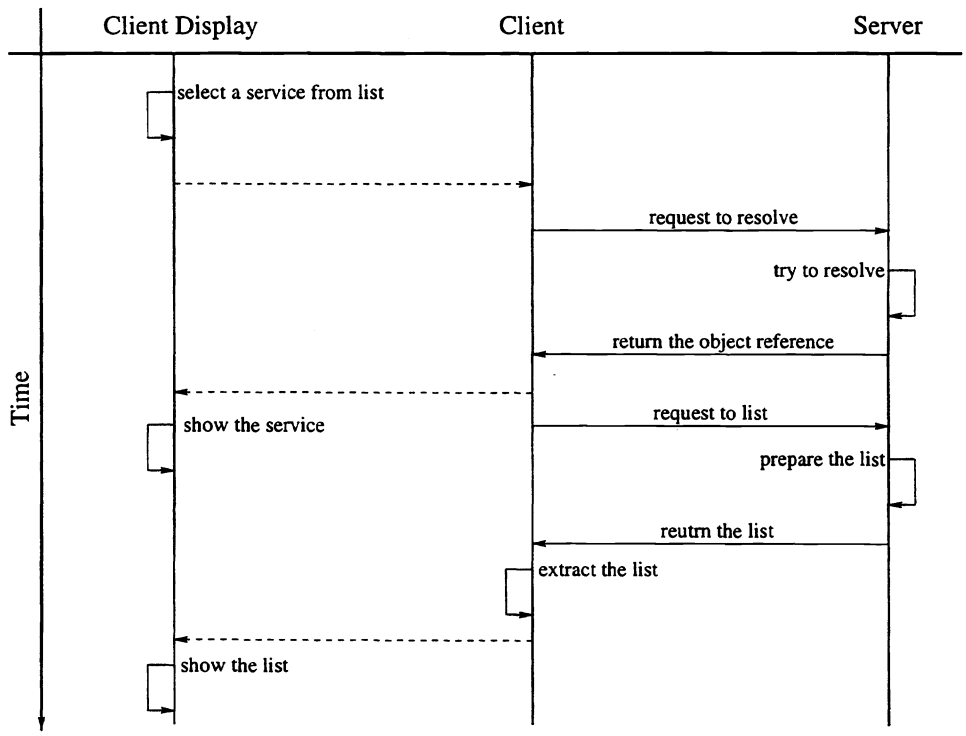

Figure 6: Interactions of Invoking a Service

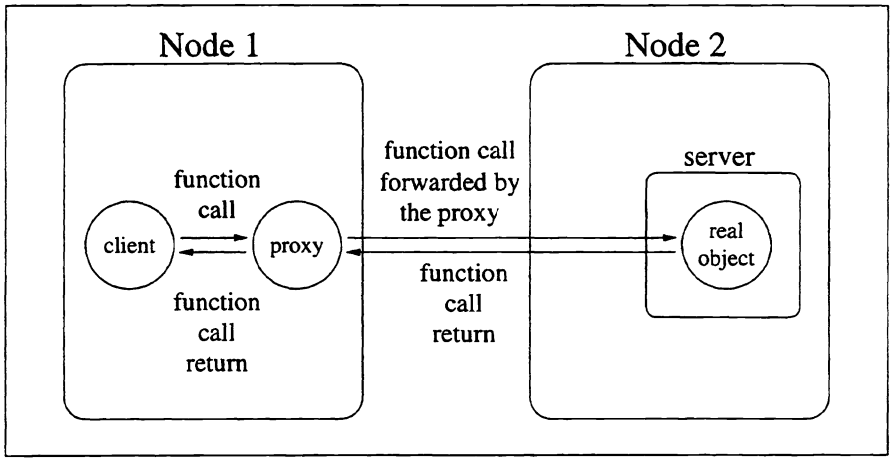

Figure 7: An Operation Call on a Proxy[3]

(1) **TIE Approach** is suitable to wrap an existing C++ class up into an interface-compatible IDL class.

(2) **BOAImpl Approach** is that the implementation class is derived from the IDL C++ class.

5.2.2. Writing a Server Program After implementing a server class, we need to write a program to inform the Orbix daemon the procedure of activating the server object. This program must have the following instructions in order:

(1) Create a new implementation object.

(2) Use CORBA::Orbix.impl_is_ready() to inform the daemon that the object is ready.

(3) Release the object.

5.2.3. Writing a Client Program The following piece of code illustrates how a client program contacts the host to get the File service:

```
DSM::File *f = DSM::File::_bind(":DSM_File", host);
f->Read(offset, length, reliable, rData);
f->_release();
```

To contact to a specific service, use the specific _bind() operation. The "::" modifier identifies the domain of the service. After all the operations done, the client program should release the object reference.

5.3. NamingContext and Service Gateway

One of the most important services a server should provide is the browsing function. In DAVIC, the browsing-related service is the NamingContext service, which is defined by OMG and adopted by DSM-CC. A NamingContext represents a node in the service domain. All the services, including NamingContext, are arranged in a tree structure. The NamingContext interface allows an administrator to insert new services into the tree, and allows an user to get a list of the available services or to resolve an object reference for the user.

5.3.1. The Service Directory The Implementation Repository (IR) in Orbix may be tree-structured, similar to the directories in a file system. Therefore, the full server name of an object consists of the path in the Implementation Repository and the interface name. How to implement the whole service tree on a Sun workstation is one of the main problems of building the server. There are two approaches to arrange the service data: (1) Database approach, and (2) File system approach. The first approach is more flexible than the second. The second is a low-cost

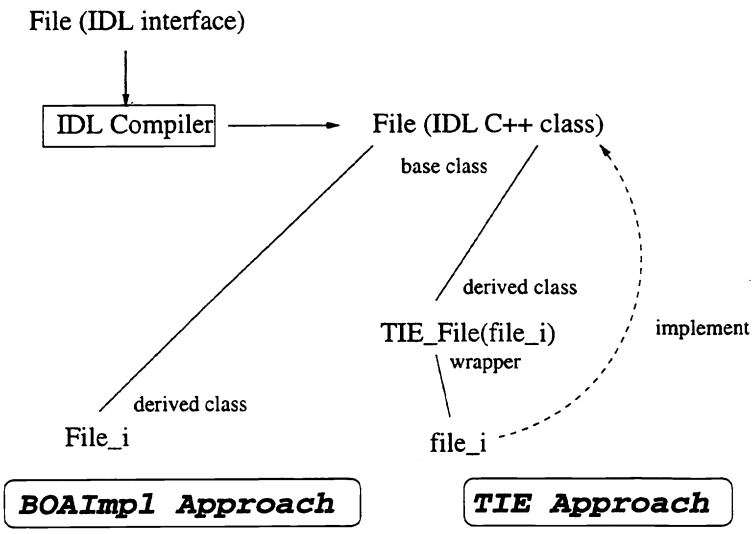

Figure 8: Relationships among IDL and related classes

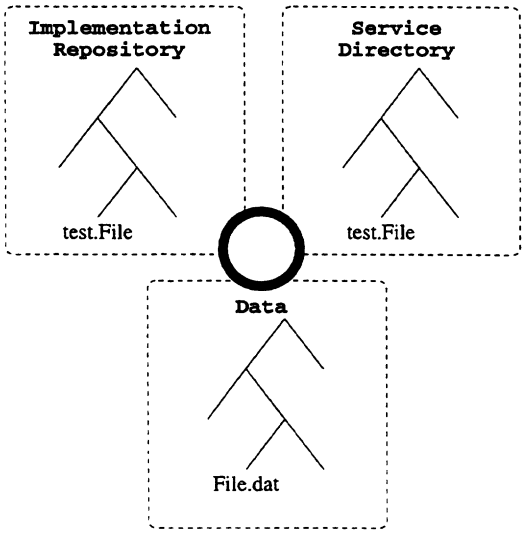

Figure 9: Relationships among Repository, Service Directory, and Data

approach when we build a simple service directory and do not care for the efficiency problem. However, we do not have a professional database management system on our workstations. Although the dbm library installed on our workstations is capable of performing relational database operations, its performance is bounded by the file system because it builds database upon the file system. Therefore, we simply choose the file system approach.

Mapping between IR and service names is easy since they have the same structure. The remaining problem is how to relate the service and the data. Because file system approach is taken to arrange our data, identical tree structures are posed on the data, the services, and the IR as shown in Figure 9. Since we do not use database to keep track of the relations among trees, the name of the service in every tree, which is identical in three trees, is responsible for that task.

5.3.2. Resolving Objects We have discussed how to get an object reference directly from the server. In this section, we will discuss another case of getting an object reference from the other services. Most of the object references are obtained through the resolve() operation of NamingContext or Service Gateway.

Orbix assumes that the object type has been known to the user, and the user get the reference by using _bind() in a certain domain, e.g., DSM::File::_bind(). The resolve() operation returns a general type of object reference, not a reference in a specific domain. This implies that the client program should narrow down the object reference to the type it needs. In order to let the client know which type the object is, we choose to record the interface type as a part of service name (this is not needed if we use database approach).

Because there are two domains in our server, CosNaming and DSM module, the resolving process has two steps: we first check the domain and then perform the resolving process in the specific domain. All services, except for Service Gateway and Directory Service, in DSM module can be resolved via DSM::Base::_bind(), because all of them are inherited from DSM::Base.

5.4. MPEGFile Service

The MPEGFile interface is used to extract data from an MPEG file, and it offers operations capable of co-operating with VCR commands.

5.4.1. The VDOBASE Library To be able to execute VCR-like control commands, we need to find a method for accessing the random access points in an MPEG file. We develop the VDOBASE library to accomplish this goal.

Table 1: State Transition Table of a VCR

Next State \ Current State	S	P	FF	FB	Pp	FFp	FBp
Command: play	P	P	P	P	Pp	Pp	Pp
Command: stop	S	S	S	S	S	S	S
Command: pause	S	Pp	FFp	FBp	P	FF	FB
Command: ff	FF	FF	FF	FF	FFp	FFp	FFp
Command: fb	FB	FB	FB	FB	FBp	FBp	FBp

States:
S–Stop, P–Play
FF–Fast Forward, FB–Fast Backward
Pp–Play with Pause
FFp–Fast Forward with Pause
FBp–Fast Backward with Pause
Commands: ff–fast forward, bf–fast backward

As described in,[6] we know that the random access points are the starting points of every group of pictures in an MPEG file.

From experiments, we know that the program *mpeg2play*, developed by Stefan Eckart, can play MPEG files with any combinations of frames starting with intra-frame picture header, the GOP header or the sequence header. Because there is no mechanism for a VCR to specify a particular time point in a video for playback, we assume that the time stamp for a normal VCR-control is not essential.

The VDOBASE contains two elements, one is an MPEG index file generator and the other is a library providing access functions. The index file generator parses an MPEG file, records all the random access points, the lengths between consecutive points, and the independent decodable lengths starting from every random access points. The random access points are defined as follows:

- Normally the random access points are the locations of intra-frame picture start code.
- If the intra-frame is the first intra-frame in a GOP, the access point moves to the location of the GOP start code.
- If this GOP is the first GOP in a sequence, the access point moves to the location of the sequence start code.

The independently decodable length is defined to be the length between the random access point and the end of the first intra-frame.

The library we developed provides functions to manipulate an MPEG file in an indexed manner. Function "seqLength()" returns the length between two consecutive points. Function "ISize()" returns the independently decodable length. The read(b, l, $offset$) operation reads in l bytes of data located at $offset$ bytes relative to the current random access point and puts them into buffer b.

5.4.2. Using VDOBASE together with VCR Commands A VCR is viewed as a state machine, and the state transitions are the results of executing VCR commands. In each state, different VDOBASE operations should be used. These states are briefly explained below, Table 1 shows the various valid state transitions.

Stop, Fast Forward with Pause, and Fast Backward with Pause: No operation.

Play: Use seqLength() to get data length l; after extracting l bytes, move to the next index and repeat this step until hitting the end of file.

Fast Forward: Use ISize() to get data length l; after extracting l bytes, move to the next index and repeat this step until hitting the end of file.

Fast Backward: Use ISize() to get data length l; after extracting l bytes, move to the previous index and repeat this step until hitting the beginning of the file.

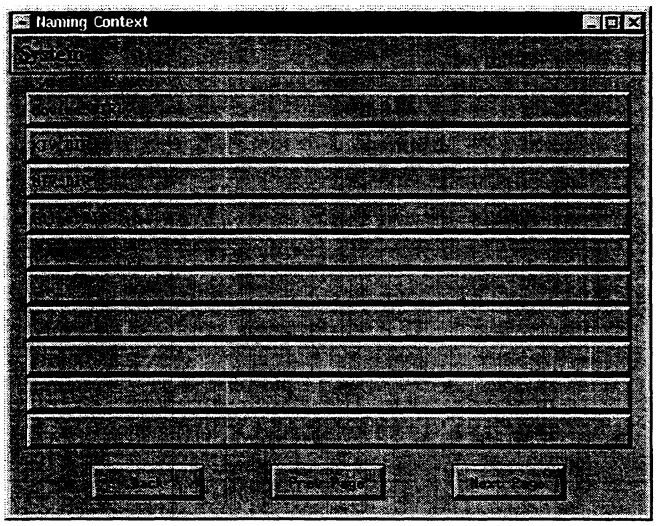

Figure 10: NamingContext Representation

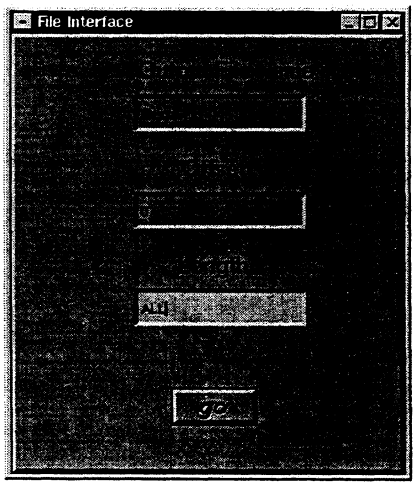

Figure 11: File Service Representation

5.5. Client Side Programs

After finishing describing all the service objects, we explain the client program in this subsection. As defined in the DAVIC specifications, the Service Gateway is the first service a client interacts. The client program first binds to ":ServiceGateway" on a host, then it browses the service directory, and requests the NamingContext to resolve new objects if further requests are issued. The task needs to be done is how to represent the services on the screen. One of our goals is that each service can run independently. To be independent of the other services, we need to use a mechanism to separate each service representation. We choose to separate each service representation by different run-time process. Though it may result in high system load, it is easy to implement our system on the SunOS.

5.5.1. Naming Context Browsing among directories requires a program to list all the available directories. The list varies every time we browse into another directory. We use Fresco to implement the browser, because it can dynamically change the attributes of a graphical object. Both Fresco and Orbix use CORBA, but some of their internal definitions conflict due to different implementation philosophy. To avoid conflicts, we split the graphical part and Orbix part into two separate processes, and these two parts communicate with each other using pipes and signals. Figure 10 shows the NamingContext representation on the screen. The user interface displays ten entries each time, and allows the user to cycle through pages to select the desired service.

5.5.2. File Service The File Service should display a form to let the user input necessary parameters. Figure 11 shows the X window interface.

5.5.3. MPEGFile Service The graphical interface of MPEGFile Service has a display window for playback and a control panel for VCR commands. Our playback program is modified from mpeg2play (developed by Stefan Eckart), because the original mpeg2play cannot read data continuously from a pipe. If the file pointer has reached the end of the MPEG file, the main program informs the control panel to enter the "stop" state. We use a signal passing from the main program to the VCR panel to indicate this situation. Figure 12 shows an active MPEGFile interface on the screen.

5.5.4. The Relationships among Various Processes Figure 13 shows the relationships among the processes on the client side. The rectangular boxes in Figure 13 represent processes which interact with an user. The oval boxes stand for the programs which contain the IDL C++ calls, and they are used for network communications to the server. The solid lines and the accompanying arrows between an oval and a rectangular boxes are pipes which are used for inter-process communications. The arrow lines between two oval boxes are the parent-child relationship. In our system, the main program of NamingContext is the parent process for all the other processes.

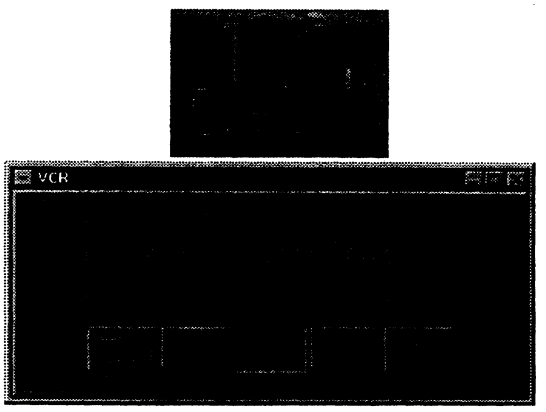

Figure 12: Playing an MPEG file

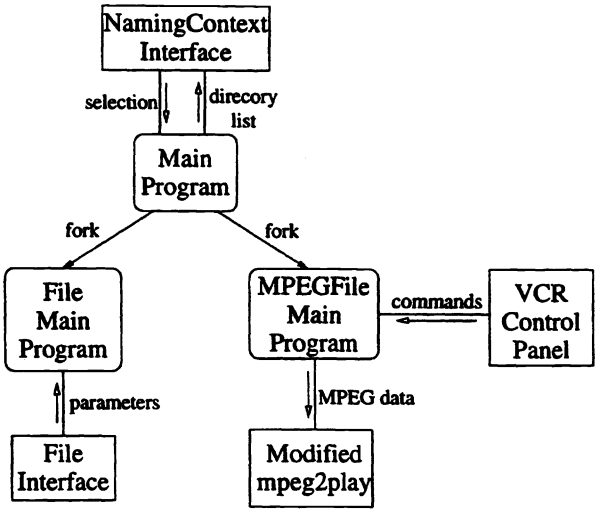

Figure 13: Relationships among Processes

6. CONCLUSIONS

In this study, we start from the DAVIC specification. In order to understand the high layer protocols in DAVIC, we study the user-to-user interfaces in DSM-CC, the CORBA architecture, and the network programming techniques. Then, we build a simple DAVIC system to demonstrate a portion of the functionalities of a DAVIC-compliant system. Throughout the implementation of our system, several programming tools are used. In summary, this system is able to offer the functions of browsing service directories, down-loading a segment of a file, and playing an MPEG video stream with VCR-like control. A simple client program is also constructed to demonstrate the high layer protocol between the server and the client. Paricularly, through the use of the VDOBASE library, MPEGFile service is capable of performing VCR-like operations.

ACKNOWLEDGMENT

The authors would like to express their gratitude to Mr. Sun-Fan Yang for his help in building VDOBASE. This work was supported in part by a grant from Computer and Communication Laboratories, Industrial Technology Research Institute (Hsinchu, Taiwan, ROC).

7. REFERENCES

[1] Digital Audio Visual Council, *DAVIC 1.0 Specifications revision 4.0*, December 1995.

[2] Object Management Group, *The Common Object Request Broker: Architecture and Specification.* Revision 1.2, December 1993.

[3] IONA Technologies Ltd., *Orbix Programmer's Guide.* Release 1.3, July 1995.

[4] Silicon Graphics, Inc. and Fujitsu, Ltd., *Fresco Reference Manual*, July 13, 1995.

[5] T.C. Zhao and M. Overmars, *Forms Library, A Graphical User Interface Toolkit for X*, 1995.

[6] ISO/IEC JTC1/SC2/WG11, *Coding of Moving Pictures and Associated Audio for Digital Storage Media at Up to about 1.5 Mbits/s*, Nov. 20, 1991.