95-AO

An Architecture for Operating System Support of Distributed Multimedia Systems*

Ya Xu, Cyril Orji, Yi Deng and Naphtali Rishe
High Performance Database Research Center
School of Computer Science
Florida International University
Miami, FL. 33199

Abstract

Distributed multimedia applications introduce new design challenges at all systems levels from network protocols and operating systems to application support platforms. This paper describes an object-oriented architecture integrating the network services with operating system to support distributed multimedia systems. The architecture, called Dynamic Object Architecture (DOA), is based on Open Distributed Processing (ODP), the international standard on distributed system and client-server architecture. DOA utilizes object-oriented technology to address the requirements of distributed multimedia systems such as continuous media, natural synchronization, dynamic Quality of Service (QoS) and group communication. The architecture also provides a path using mature industry standards to develop new applications while retaining compatibility of old applications.

1 Introduction

Distributed multimedia applications introduce new design challenges at all systems levels from network protocols and operating systems to application support platforms. Early multimedia systems (MMSs) provided a hardware front-end to support the transmission and presentation of different media types such as video and audio. However, it is now recognized that this is not sufficient and that a class of applications that requires direct access to continuous media data types exists. It has always been recognized that the operating system was needed to facilitate multimedia applications, and in addition it was also recognized that micro-kernels, user-level threads and split level scheduling have important roles to play in supporting continuous media. Little research, however, was done on integrating network services with operating system functionality to support Distributed Multimedia Systems (DMSs). The main goal of this integration is to retain the transparency between the network communication protocols and application programs thereby allowing programmers to use familiar concepts while invoking remote operations.

MMSs have certain characteristics which existing supports in traditional systems are illequipped to address. These include continuous media, natural synchronization, dynamic Quality of Service (QoS) and group communications [2]. For example, due to the continuous nature of multimedia data, caching cannot be effectively used to improve data access rate in MMSs. Moreover, this continuous nature of the data makes the static semantics of the traditional remote procedure call (RPC) inappropriate in MMSs. Although parallel I/O techniques have been effectively used to improve I/O rate in traditional systems, the synchronization delay requirements of multimedia applications introduce another dimension to the problem. Specifically, sets of real-time presentation devices in multimedia systems must be tied together so that they consume data in fixed ratios even when their incoming data originate from different sources. While data transmission in traditional applications emphasizes only data reliability, the synchronization delay requirements of multimedia systems require data transmission not only to be reliable but also delay-sensitive.

The OSI reference model and protocol also exhibit certain limitations to multimedia applications. In particular, in traditional applications, the value of the QoS parameter is static during the lifetime of a connection. However, in multimedia applications, it is desirable to be able to re-negotiate the value of a QoS parameter at runtime [1]. This cannot be done with the current OSI protocols. Moreover, the point-to-point characteristics of the OSI reference model also make it unsuitable for group communication [15]. Group communication – a typical multimedia application, is typified by multimedia conference.

Distributed multimedia environments will generally be heterogeneous, consisting of many different workstations interconnected by one or more types of networks. With this inherent heterogeneity, it is important that DMSs are open. The guarantees of open property need interconnectivity, interoperability and portability. Although client-server distributed systems support a level of interoperability, experience with such systems has been predominantly with local area networks (LANs). The basic client-server model is unlikely to provide the total solution for DMSs

*This work has been supported in part by g rants from NASA ⁻F.

because of the complexity of migrating from locally distributed systems to more global systems [9]. The object-oriented approach shows promise in addressing this complexity. Therefore, we adopt standardization work of Open Distributed Processing (ODP) of ISO and use the encapsulation and inheritance property of object orientation to increase interoperability.

In this paper we develop the Dynamic Object Architecture (DOA) as a framework for integrating network services with operating system. This architecture is based on Open Distributed Processing (ODP), the international standard on distributed system and client-server architecture. The DOA utilizes object-oriented technology to address new requirements in distributed multimedia systems such as continuous media, natural synchronization, dynamic QoS and group communication. The architecture also provides a path whereby well established industry standards can be used to develop new applications facilitating compatibility with old applications.

The remainder of the paper is organized as follows. Section 2 surveys related work in this area. Section 3 presents a brief overview of the OSI and ODP standards. Section 4 presents our Dynamic Object Architecture (DOA) based on ODP Reference Model and object-oriented technology. Section 5 discusses the implementation of key components of the DOA. We conclude the paper in Section 6 briefly noting on-going work.

2 Related Work

In this section we review some previous work in this area. Research in operating system support for multimedia applications has so far fallen into two broad categories. In the first category, effort was directed mostly in building custom software running on specialized hardware to support multimedia applications. Typical efforts in this direction include the Pandora system [5], the Pegasus project [6] and the IBM HeiTS system [4]. In the second category, existing operating systems are modified to include support for multimedia applications. Examples include work on the UNIX SVR4 scheduler [8], extensions to the Chorus micro-kernel [3] and thread implementation in the ARTS operating system [11].

Pandora [5], an experimental system for networked multimedia applications, uses a sub-system to handle the multimedia peripherals. It uses transputers and associated Occam code to implement the time critical functions. Stream implementation is based on self-contained segments of data containing information for delivery, synchronization and error recovery. Buffer allocation scheme allows for the transport of audio and video format data. This is achieved by using two specialized types of buffers: decoupling buffers between processes or hardware units that do not run synchronously, and clawback buffers to enable streams with jitter to be synchronized with the local clock.

In the Pegasus Project [6], an attempt is made to design and implement a general-purpose operating system to support distributed multimedia applications. One of the primary goals of this project was to facilitate user-level interactive processing of multimedia data while at the same time maintaining all the desirable properties of a distributed system such as resource sharing, data sharing, security, and fault tolerance. Pegasus uses a shared address space for local groups of mutually trusted machines that share the same data representation. Object storage is tailored to efficient management of persistent objects and multimedia data, and the file system is log-structured.

IBM has developed a new-generation end-to-end communication system called HeiTS [4]. HeiTS is designed to handle high-speed data applications as well as multimedia applications within IBM's Small Systems line (PS/2 under OS/2 and the RISC System/6000 under AIX). Two of the many attractive features in HeiTS are the satisfaction of real-time requirements and efficient data handling capability. HeiTS uses threads to handle audiovisual data streams with real-time requirements. A Resource Management System has been implemented in HeiTS to support this kind of scheduling. It allows best effort and guaranteed connections, and supplies the scheduler with the necessary information for real-time scheduling. With respect to efficient data handling, a high performance Buffer Management System has been implemented which supports segmenting and recombining of data units, chaining and locking of buffers. The net effect of these features is reduced overhead and the reduction of many unnecessary data movements in the system. HeiTS also implements the lower four layers of the OSI Reference Model that allows multicast on the network layer, multiplexing up to the data link layer, segmentation, and end-to-end flow control.

In [8] an approach on the use of existing operating systems for the processing of continuous media data is provided. It is shown that existing scheduler in UNIX SVR4 is unacceptable when dealing with continuous media applications. A new scheduling class for SVR4 that provides significant improvements in performance over the existing UNIX SVR4 scheduler is proposed and analyzed.

A micro-kernel based approach for dealing with the requirements of continuous media has also been proposed [3]. Specifically, in [3], extending the Chorus micro-kernel architecture to support end-to-end quality of service (QoS) was proposed. The key concept deals with representing QoS controlled communication between user level threads on potentially different machines, a split level scheduling architecture and a rate-based transport protocol.

An implementation of user level threads in the ARTS operating system is discussed in [11]. Two types of threads – periodic and aperiodic threads are described. Periodic threads, are defined by start time, period, deadline and worst case execution time, while aperiodic threads are defined by deadline, worst case execution time and worst case interval time. ARTS supports a split level scheduling scheme where a user level scheduler manages user level threads while a meta-level scheduler takes a global view across all processes. A deadline handler can also be defined on a thread-to-thread basis to manage quality of service degradation. These works demonstrate that the use of micro-kernel, user-level threads and split level scheduling have important roles to play in supporting continuous media. However, considerable work is required on integrating operating system functionalities with network services.

Other related work in DMSs has been in the area of communications and networking [12]. On end-system architectures, the work in [10, 7, 17] are rather too abstract to represent a practical end system. Moreover, by assuming the basic ISO/OSI model and not suggesting extensions to it, these various research efforts were limited in their abilities to meet the new requirements of DMSs.

3 OSI and ODP standards

Before we begin to discuss the DOA architecture, first let us review the OSI and ODP standards. The ISO OSI [1] provides a framework for communcation protocols [16]. It organizes the protocols as seven layers and specifies the functions of each layer with user programs running on the application layer.

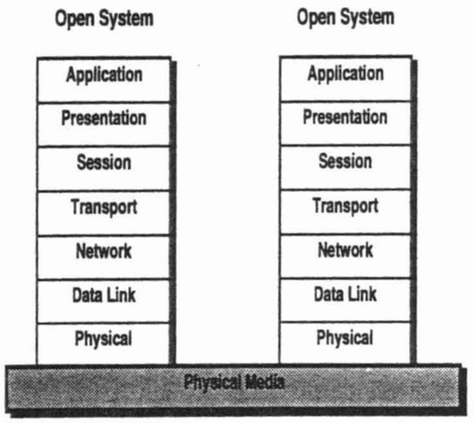

Figure 1: **OSI Reference Model**

Although a detailed description of the OSI-RM is beyond the scope of this paper (see [16, 13]), we briefly highlight the purpose of each layer in the model. Layer 1 or the *physical layer* hides the nature of the physical media from the data link layer to maximize the transportability of higher layer protocols. Layer 2 or the *data link layer* is responsible for error free data transmission over a data link. Layer 3, the *network layer*, provides interconnection services. It provides transparency over the topology of the network as well as transparency over the transmission media used in each sub-network comprising the network. Layer 4, the *transport layer*, is responsible for moving data reliably from one end system to another end system. While the end-to-end service provided by the transport layer deals with data transfer between the end systems, the three topmost layers (session, presentation and application) provide an inter-working service. Layer 5, the *session layer* is primarily responsible for the coordination function, while layer 6, the *presentation layer* is responsible for the representation function. Layer 7 or the *application layer* provides the rest of the communication functions that may be specific or generic to a class of applications.

[1]International Standards Organization Reference Model of Open Systems Interconnection

As already noted, the standards to achieve interoperability include communication and non-communication standards. ODP is the evolving non-communication standards that addresses distributed processing in an open system environment.

ODP is the result of a joint effort by ISO and CCITT to develop uniform standards across multiple systems and their components. The initial goal of ODP is a reference model to integrate a wide range of future ODP standards for distributed systems and to maintain consistency across such systems, despite heterogeneity in hardware, operating system, networks, programming languages, databases, and management authorities [9].

The ODP Reference Model(ODP-RM) [14] serves to:

- model distributed processing in terms of functional components,
- identify levels of abstractions at which services can be described,
- classify the boundaries between components,
- identify the generic functions performed by distributed systems, and
- show how the elements of the model can be combined to achieve ODP.

The ODP standard identifies seven different aspects of an ODP system. Each aspect is a logical grouping of the functional requirements of a distributed system. These seven aspects are storage, process, user access, separation, identification, management, and security. Each aspect can be viewed in five different ways. These five viewpoints are enterprise, information, computational, engineering, and technology viewpoints [9]. Each viewpoint leads to a representation or an abstraction of an aspect of the system with emphasis on a particular set of concerns. The enterprise viewpoint is concerned with the social, managerial, financial, and legal policy issues that constrain the human and machine roles of a distributed system and its environment. The information viewpoint concentrates on information modeling and flow, plus structure and information manipulation constraints. The computational viewpoint focuses on the structure of application components and the exchange of data and control among them. The engineering viewpoint concerns the mechanisms that provide the distribution transparencies to the application components. The technology viewpoint focuses on the constraints imposed by technology and the components from which the distributed system is constructed.

Our goal is to integrate the network services with operating system to support distributed multimedia systems. The most important requirement is transparency. Moreover, we are concerned about interoperability and portability from the viewpoint of operating system support that is end system-related, not communication-related. Given these requirements and some of the deficiencies of the OSI model with respect to multimedia applications (see Section 1), we adopt the ODP as the appropriate model to address these problems.

4 Dynamic Object Architecture (DOA)

In this section we describe the dynamic object architecture (DOA) and show its relationship to the reference model of open distributed processing (RM-ODP).

The DOA is a layered architecture for integrating network services with operating system in order to support DMSs. It supports mechanisms that hide the underlying system's heterogeneity from users and applications. These mechanisms not only address such general issues on network services as access, location, migration, concurrence, failure, and transparency, but also support the characteristics of multimedia applications, such as continuous media, natural synchronization, dynamic QoS and group communication. The most fundamental architectural concept that we use is the notion of dynamic object. The dynamic object utilizes the object-oriented technology and provides the network services with dynamic functionality and semantics to meet the new requirements of DMSs.

The DOA is constructed fully according to the ODP system's general architecture. Because ODP is an international standard on distributed systems, the DOA which is based on ODP appropriately reflects the nature of distributed applications and maintains consistency across systems, despite heterogeneity in hardware, operating system, networks, programming languages, databases, and management authorities. Therefore, DOA integrates distribution, interoperability and portability and provides an open infrastructure for DMSs. The DOA consists of four object layers as shown in Figure 2. These are

- the computational object,
- the engineering object,
- the transparency object and
- the nucleus object layers.

4.1 The computational object layer

The computational object layer specifies the computational structures and statements of properties for interaction between objects. It focuses on the structure of application components and the exchange of data and control among them. This is a typical application platform based on client/server model.

The computational object layer includes client and object entities. A client is an entity that wishes to invoke an operation on a target object entity. An object is an identifiable, encapsulated entity providing one or more services that a client can request. A client can identify the object and knows the services that the object can provide, but it can not access the internal structure of an object. An object can be created and destroyed as a result of executing object requests. For example, in a multimedia conference, when a person joins the conference, an object entity is created. When the person exits from the conference, the corresponding object entity is destroyed.

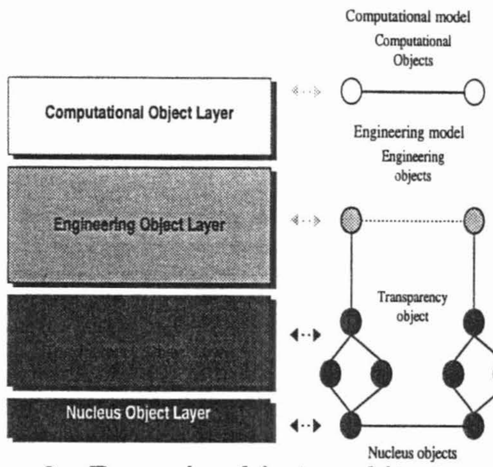

Figure 2: **Dynamic object architecture mapping to open distributed processing reference model.** *On the left is shown the dynamic object architecture (DOA) and on the right is the corresponding reference model of the open distributed processing (RM-ODP).*

4.2 The engineering object layer

The engineering object layer focuses on the mechanisms to assure the realization of properties of application components whose structures were defined in the computational object layer. In the engineering object layer, the components of a client entity include the dynamic object interface (DOI), the interface repository (IR) and the interface definition language (IDL), while the components of an object entity include the IDL and the object adapter (OA).

4.2.1 The dynamic object interface (DOI)

A client program uses DOI to name the request's target object and calls on the Object Communication Support (OCS) services to add the required arguments to the request. When a client program invokes an operation on an object, the corresponding DOI for the target object is called. The DOI is responsible for organizing the information required to execute the operation before calling on a transport mechanism, such as RPC, Socket, TLI, or NetBIOS, to deliver the request to the target object for execution. In a traditional call, each DOI object corresponds to a particular operation on a particular object. We call it the static call interface.

Because of the dynamic requirements of multimedia communication, the static call interface is ill-equipped to handle multimedia communication. For example, in a multimedia conference, the requirements on system resources are dynamic. This could be a result of members joining or leaving a conference in session. A result of this could be that the quality of service (QoS) negotiated at the start of the conference is no longer appropriate. There is, therefore, a need to re-negotiate the QoS. Traditionally, this would be handled by terminating the current session and starting a completely new session. This technique is clearly inappropriate, hence the need to be able to dynamically re-negotiate the QoS in a manner that is transparent to the parties in the on-going conference.

There are also other motivations for a dynamic interface in multimedia applications. Consider for example a typical *getvideo()* function with a static interface that retrieves a video program on demand. Two major problems with this type of static interface will be:

1. A bulk of video data will be transferred to client at one time saturating the network.

2. The synchronization between intra-media and inter-media is impossible.

A dynamic interface will solve these problems. In our DOA, the DOI is used to realize dynamic interface. Its main functions are two fold:

1. *Interface reconstruction*: The client call is reconstructed to satisfy the dynamic semantics. For example, the GetVideo is, typically, added with required limitation on media synchronization. The process is hidden from the client. Once it constructs the new request, the OCS delivers it to an object adapter that parses the request before arranging for its execution.

2. *Interface inheritance*: This is used for traditional data communication interface and fixed multimedia application interface. Typically, the interface for data communication can be static. Thus, if we treat all interface calls as dynamic calls, the efficiency of the interface would degrade because the overhead of a dynamic call is clearly larger than that of a static call. In other words, it is desirable to make dynamic calls only where needed. We use interface inheritance to directly utilize an existing interface instead of reconstructing the request to object entity. On the other hand, we may also save some calls on multimedia communication in the interface repository thereby improving system efficiency. This is done by using inheritance on a current interface instead of constructing a new interface.

4.2.2 Interface repository

The Interface repository supports the DOI by storing objects representing IDL information in a form used at runtime. On receipt of an application's request, a client typically interrogates the interface repository by the DOI to determine the interfaces capable of satisfying the request. If needed, the client may use dynamic invocation interface primitives to construct the argument list of a request to the selected target object. Once it constructs the request, the communication support delivers it to an object adapter that parses the request before arranging for its execution. The client can call the interface repository directly and decide the interface satisfying the requirement. Then it uses DOI primitives to construct the request.

4.2.3 Interface definition language (IDL)

The IDL describes the operations and associated attributes of an object interface in terms that the rest of the system can understand. IDL also makes it possible to translate the functionality offered by resources into object-oriented interface. In fact, it is used to define DOI and provide the information that existing programming languages do not provide. From the IDL, DOI and object adapter can be generated automatically by an IDL compiler. The IDL is derived from C++ and adds extra information including the direction in which parameters travel, discriminators and so on.

4.2.4 Object adapter

The distributed multimedia systems make it possible that the synthesis of existing objects exists in the whole system. These object entities can have different constructions. The object adapter provides the object communication platform for portable object implementations.

Object adapters serve a dual purpose. First, they provide the main interface through which object implementations invoke the object communication support services. Second, they augment the basic object communication support model by implementing support for richer object-modeling features. Furthermore, object adapters provide a generic interface for all object entities. The generic interface supports reference for new object entities while providing compatibility for old object entities. Through inheritance, an object adapter can be extended to a library of object adapters to support different object entities. Figure 3 shows the components and interface of the DOA. Note in particular the callback call interface from object adapter to object entity. This is similar to the callback function in X/Window.

4.3 The transparency object layer

A computational object can call on a number of transparency objects of the transparency object layer. Each transparency object represents a system property required to realize distribution transparency. Transparency objects at each end-system require the services of nucleus object layer, an abstraction of the local host environment, and the communication services necessary for inter-nucleus interactions.

The transparency object layer includes an OCS. The OCS provides services to deliver requests between clients and objects. These services include re-

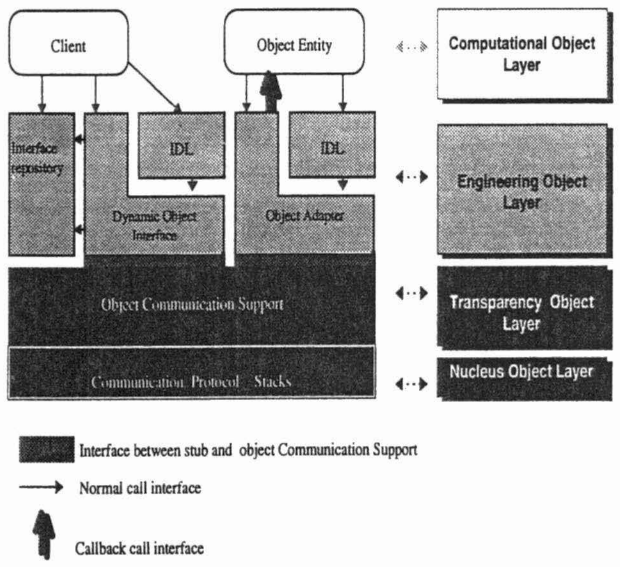

Figure 3: **The components and interface of DOA.**

source negotiation, object location, message delivery and method binding. Here, method binding means that a virtual call is bound to multimedia semantics with concrete network connections. On the other hand, the host environment and communication protocols of end-systems may be different because of the nature of the distributed system. OCS provides interfaces that mask differences between the mechanisms found across different end-systems. While the dynamic object interface provides a client with multimedia semantic interface, the OCS is fully transparent to the client.

4.4 The nucleus object layer

The nucleus object layer includes the communication protocol stacks. The communication protocol stacks provide network services depending on end-system. The transport layer which is the top layer of the communication protocol stacks, provide network and communication services to the OCS in the transparency object layer. The reason is that many heterogeneous systems could be interconnected by TCP/IP that typically provides the transport layer services. This will facilitate the use of de facto industry standards in the development of new services to meet the requirements of DMSs. The service interfaces that the communication protocol stacks provide include RPC, Socket, TLI or NetBIOS. These widely used industry standards facilitate compatibility between new applications and old applications. Moreover, they also provide a path of smooth transition from traditional data communication services to new DMSs services.

5 Implementation of DOA

In this section we present implementation details of the dynamic object architecture (DOA). We use object-oriented technology and C++ in our implementation. The key components in the DOA are the dynamic object interface (DOI) and the object communication support.

5.1 Dynamic object interface

The DOI encapsulates the request to object from the client. Its basic structure is shown below using C++:

```
class doi { // doi class
public: // doi interfaces
        doi  *registerOE();
// register  the service to object entity
void   callOE();
// request the service to object entity
...   ...   ...   ...   ...   ...   ...   ...
protected: // doi parameters and attribute
qostruct  qosdata ; // QoS requirement
Oeid  id; // ID of object entity
BufID  Rbuffer;
// The pointer to receiver buffer
BufID  Sbuffer ;
// The pointer to sender buffer
...   ...   ...   ...   ...   ...   ...   ...
private:
listDOI *doistruc;
// dynamic list of doi structure
}
```

In the **doi** class, two main services are provided. The first is *registerOE*, which is used for registration for object entity services and the second is *callOE*, which is used for requesting object entity services.

RegisterOE() registers the service using parameter including object entity information, such as port, host name, and QoS requirements. The register process might include the QoS negotiation according to the system resources. It typically returns a new doi object pointer and records the new object pointer in the listDOI of old doi object.

Because of the natural synchronization and group communication of multimedia, the semantics of request might include mutiple requests to multiple objects, distributed in different location. For example, the retrieve of multimedia database will include video frames, audio samples and text pieces. It is not appropriate to transfer all these in a single communication channel because they have different QoS requirement. If the best QoS is selected as the QoS of communication channel, system resources would be wasted. Conversely, if the worst QoS is selected as the QoS of the communication channel, media data requiring higher QoS would be greatly affected. Moreover, if the information is distributed in different locations, it is not possible to transfer them in the same channel. Another example is the group communication in multimedia conference where a request is typically sent to multiple sites. Using a simple channel and single QoS to address the application would be very difficult. Therefore, in the DOI, every media registers its own object entity request according to its own QoS requirements and object entity. A new doi object is created dynamically to record the information about the request. The old doi object maintains a dynamic list of new doi objects to keep track of the entire call status and for synchronization.

The structure of a dynamic **doi** object list is:

```
struct  listDOI{
doi * dp; // the pointer to old doi
listDOI* next; // next pointer
}
```

The *callOE* function is polymorphic. It can call different procedures according to different media requirements. On the other hand, callOE() can inherit interface from run-time interface repository. Typically, it inherits the interface on traditional data communication. This helps it to avoid much work after registration because a new object would have to be created to construct the session between client and object entity after registration. This process is complex, and to improve efficiency, some fixed interfaces may be stored in the interface repository so that callOE() utilize them using inheritance.

The data buffer used for storing user data for transmission is also dynamically used. After registration, a client gets a pointer to the data buffer. The size of buffer depends on the media type. As discussed above, each media communication uses a different communication channel corresponding to a different **doi** object. Therefore, the intermedia synchronization needs to be completed by the client while intramedia synchronization is completed by the DOI and object communication support. Inter-media synchronization is required where some temporal relationship exists between two objects in a multimedia object, such as lip-sync between an object of type audio and an object of type video. We consider real-time data like audio and video as these have well-defined intermedia temporal relationships. Transmission of such data generates periodic, asynchronous data streams.

The data buffer size, is organized according to the requirements of inter-media synchronization. The time synchronization does not mean that every stream has the same number of bytes. It is apparent that video frames and audio samples have different byte sizes and are synchronized according to the number of video frames and number of audio samples.

The DOI utilizes the object-oriented mechanism to realize the support of dynamic characteristics of DMSs. Meanwhile, in order to improve efficiency, we utilize inheritance to realize the interface repository. In a sense, this is, in fact, a static interface. DOA combines the static interface with dynamic interface to provide services for new application. The inheritance keeps the compatibility to the old application besides improving efficiency.

5.2 Object communication support

When the interface is constructed, the objects of object communication support need to be created to provide services for DOI. We define the OCS class as follows:

```
class ocs{
public:
void setqos(); // negotiating QoS parameters
void callOCS();
// request services of underlying layer
...   ...   ...   ...   ...   ...   ...   ...
private:
void adjustqos();
// adjusting QoS parameters
qostruct qosdata; // QoS parameters
BufID *Sbuff; // send buffer area
BufID *Rbuff; // receive buffer area
...   ...   ...   ...   ...   ...   ...   ...
}
```

Two important functions are realized in OCS. These are intra-media synchronization and dynamic QoS negotiation. Intra-media synchronization is concerned with delivering each object in time to meet the respective playout deadline. When the request arrives at the OCS through DOI, the OCS object automatically realizes the intra-media synchronization. A step toward achieving this goal is to generate a transmission schedule by deriving transmission deadlines from the player deadlines after taking network and other delay into account. To compensate for anomalies, appropriate handling schemes are required at the OCS object. Intra-stream synchronization also requires calculation of maximum buffer sizes needed for each stream, prior to the transmission. This is completed by QoS negotiation. Buffers serve to synchronize the stream data by smoothing out network delays and jitter in the individual streams. The intra-stream synchronization is a very low level synchronization that deals with maintaining synchronization of individual data streams originating from communication channels. The high level synchronization is completed by the client program as already discussed.

The dynamic QoS negotiation occurs during the communication. In the beginning, the client procedure calls DOI to negotiate the QoS parameter. The parameter is typically defined as a tuple that includes speed ratio, utilization, average delay, maximum jitter, maximum Bit Error Rate (BER) and maximum Packet Error Rate (PER). The reliability, expressed in terms of BER and PER, represents the number of errors per time unit for bits and packets, respectively. Each OCS object uses the QoS parameters to establish the communication channel for DOI object. During communication, however, the change of resource might lead to a re-negotiation of QoS value. A new QoS tuple should be set up according to the resource utilization. The OCS object selects a temporal synchronization mechanism to mask the process of changing the value of QoS. After QoS is re-negotiated, the OCS stops the temporal synchronization mechanism and returns to normal communication status. The client procedure is unaffected by the adjusting of QoS value.

6 Conclusion and Future Work

We have presented a novel architecture – the *Dynamic Object Architecture (DOA)* for integrating network services with operating system to support distributed multimedia systems. This architecture is based on the *Open Distributed Processing Reference Model (ODP-RM)*. The architecture ensures transparency of applications to networks and allows for development of new applications.

This research is ongoing, and is actually a part of a broader research to study multimedia applications under a heterogeneous, distributed environment. The research is currently being done under two major computer platforms - personal computers and Sun workstations. When different operating systems, hardware architectures and network protocols co-exist, it becomes necessary to integrate network services with operating system so that the complex network communication is transparent to applications. However, because the traditional RPC protocol does not meet the new requirements of distributed multimedia applications, and because the static interface and semantics for traditional data transmission do not meet the dynamic property of multimedia applications, there is a need for an architectural framework to address these problems. Hence our design of a new architecture for the distributed multimedia application.

References

[1] A. Campbell, G. Coulson, and D. Hutchison. A Quality of Service Architecture. *ACM Computer Communication Review*, pages 6–27, 1994.

[2] G. Coulson, G. Blair, P. Robin, and D. Shepherd. Extensions to ANSA for multimedia computing. *Computer Network and ISDN Systems*, pages 306–323, 1992.

[3] G. Coulson, G. Blair, P. Robin, and D. Shepherd. Extending the Chorus Micro-Kernel to Support Continuous Media Applications. *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 49–60, 1993.

[4] D. Hehmann, R. Herrtwich, W. Schulz, T. Schutt, and R. Steinmetz. Implementating HeiTS: Architecture and Implementation Strategy of the Heidelberg High-Speed Transport System. 1993.

[5] A. Jones and A. Hopper. Handling Audio and Video Streams in a Distributed Environment. *Proceedings of ACM SIGOPS'93*, pages 231–243, 12 1993.

[6] I. Leslie, D. McAuley, and S. Mullender. Pegasus - Operating System Support for Distributed Multimedia Systems. pages 69–78, 11 1992.

[7] T. Little and A. Ghafoor. Network Considerations for Distributed Multimedia Object Composition and Communication. *IEEE Network Magazine*, pages 32–49, nov 1990.

[8] Neih, J. Hanko, D. Northcutt, and G. Wall. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 35–47, 1993.

[9] J. Nicol, T. Wilkes, and F. Manola. Object Orientation in Heterogeneous Distributed Computing Systems. *IEEE Computer*, pages 57–67, June 1993.

[10] C. Nicolaou. An Architecture for Real-time Multimedia Communication System. *IEEE Journal on Selected Areas in Communications*, 8(3):391–400, apr 1990.

[11] Oikawa and H. Tokuda. User-Level Real-Time Threads. *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 61–71, 1993.

[12] K. Ravindran and V. Bansal. Delay Compensation Protocols for Synchronization of Multimedia Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 5(4):574–589, aug 1993.

[13] A. Tanenbaum. Network Protocols. *ACM Computing Surveys*, 13(4):453–489, December 1981.

[14] A. Tang and S. Scoggins. *Open Networking with OSI*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[15] M. Woo, N. Qazi, and A. Ghafoor. A Synchronization Framework for Communication of Pre-orchestrated Multimedia Information. *IEEE Network Magazine*, pages 52–61, January 1994.

[16] H. Zimmermann. OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications, Com-28*, pages 425–432, April 1980.

[17] T. Znati, Y. Deng, B. Field, and S. Chang. A Multilevel Specification and Protocol Simulation Tool for Distributed Multimedia Communication. *International Journal in Computer Simulation*, pages 355–382, 1993.