Supporting Advanced Multimedia Telecommunications Services Using the Distributed Component Object Model

Dionisis X. Adamopoulos[1], Prof. George Pavlou[1], Dr. Constantine A. Papandreou[2]

[1] Centre for Communication Systems Research, University of Surrey, UK
{D.Adamopoulos, G.Pavlou}@ee.surrey.ac.uk
[2] Hellenic Telecommunications Organisation (OTE), Greece
kospap@org.ote.gr

Abstract. The demand for a great variety of sophisticated telecommunications services with multimedia characteristics is increasing. This trend highlights the need for the efficient creation of distributed programs with multimedia data exchanges over distributed processing environments. Therefore, it is necessary to support the object-oriented development of distributed multimedia applications in a flexible manner. This paper recognises Microsoft's Distributed Component Object Model (DCOM) as a key potential technology in the area of service engineering and examines a structured approach to enhance it for the handling of continuous media streams through the design and implementation of a collection of suitable multimedia support services. The proposed approach focuses on the modelling of continuous media communications in DCOM and is validated through the design and implementation of a multimedia conferencing service.

1 Introduction

Driven by technological advances, market growth and deregulation, the global telecommunications industry is rapidly adopting a highly dynamic and open character, which, in combination with the evolving synergy between information and telecommunication technologies, provides a wide range of opportunities for the delivery of advanced multimedia telecommunications services (also referred to as *telematic* services). Based on recent developments in object orientation and distributed computing, these telecommunications services are designed, realised and deployed as multimedia applications operating on distributed computing platforms [1][16]; the latter are also known as Distributed Processing Environments (DPEs).

Despite the fact that multimedia support has been considered in general terms in the ISO/ITU-T Reference Model for Open Distributed Processing (RM-ODP) [7][12], it has not yet been considered in Microsoft's Distributed Component Object Model (DCOM) [3], and is not yet mature in the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [11][17]. Recently, a wide range of new telecommunications services are becoming increasingly popular by employing video to convey information and to enhance communication among human users (e.g. videoconferencing, video on-demand, etc.). Therefore, in the emerging multi-vendor, multi-stakeholder telecommunications environment, it is necessary to facilitate the rapid and flexible deployment of a great diversity of multimedia, multi-party services by providing support for continuous media in DPEs. The role of DCOM is expected to be important as it is one of the promising distributed object platforms for service engineering [2]. Key advantages are its ubiquity and the fact that it supports key DPE features such as multiple interfaces per object and object groups [22].

This paper presents an approach which extends DCOM to an environment suitable for the development of advanced multimedia telecommunications services. More specifically, it examines central issues associated with the provision of object-oriented support for the handling of continuous media in terms of representation, transmission, and management. The proposed approach is validated through the design and implementation of a multimedia conferencing service. Finally, experiments are conducted in order to assess the flexibility and efficiency of the proposed approach and conclusions are drawn.

2 Modelling Multimedia Telecommunications Services

Multimedia communication involves the interaction of devices which can deal with networked suppliers and consumers of various types of digitally represented information. The tasks broadly involved in this process can be divided into the coding and transport of the different media, and into related control aspects, such as how to locate services, request transfer, establish and maintain connections, ensure integrity and timeliness, and handle presentation issues during the delivery of multimedia information. These control aspects are the concern of this paper, since they are particularly important for the realisation of the full potential of distributed object platforms [15]. Another important requirement is the ability to hide the heterogeneous low-level aspects of dealing with streams through high-level Application Programming Interfaces (APIs) and to provide abstractions which could be easily dealt with by non-network programmers. For the rest of this section, we examine other research and standardisation work related to the flexible handling of multimedia streams.

The model of object interaction conventionally adopted in distributed object platforms (i.e. remote method invocation) is inappropriate for continuous or dynamic media, i.e. media which contain a temporal element, such as real-time audio or video. For these media types, a streaming, i.e. continuous mode of interaction is required rather than a request / response method invocation model. The main difference from discrete data interaction is that continuous interaction is not atomic since it models the exchange of continuous data (an on-going communications activity) between multimedia objects [4][13].

This is in full agreement with the RM-ODP’s multimedia computational model, which uses stream interfaces to build streaming interaction over the primitive notion of a signal [7][12]. Streams are also present in several other distributed computing architectures. Initially, they appeared in the Multimedia Systems Services (MSS) architecture, which was proposed by the Interactive Multimedia Association (IMA) [9]. In this approach, the objects producing the streams are "special" and inaccessible to applications; the latter can control, but not directly access, the real-time media. The IMA MSS is currently being adopted and extended by ISO in its PREMO standard [10]. Furthermore, much of the initial work on streams, which influenced greatly the RM-ODP, took place trying to address the requirements of multimedia support in the Advanced Network Systems Architecture (ANSA) [4]. The current ANSA Phase III Distributed Interactive MultiMedia Architecture (DIMMA) project is based on the ANSAware distributed systems platform, which has been enhanced with a modular protocol stack and a flexible multiplexing structure [14].

OMG has recently addressed the need for streams and real-time services in CORBA by issuing a request for proposals (RFP) for the control and management of audio / video streams, and by summarising submissions in [17]. However, this RFP does not examine the implementation of streams in CORBA. Such implementation issues were addressed by specific Object Request Broker (ORB) vendors [11], and by the ACTS ReTINA project, which designed a distributed object platform based on CORBA, enhanced with streams and Quality of Service (QoS) extensions [5]. Another CORBA version 2.0 compliant implementation considering multimedia support is the TAO ORB, which runs on real-time operating system platforms, and is primarily designed for strict real-time applications [19].

In all the above architectures, the modelling of continuous media communications through a flexible, high-level but efficient infrastructure are crucial. More specifically, modelling mainly involves the choice of suitable and sufficient abstractions, their efficient implementation upon a target DPE, and the adoption of appropriate interaction patterns and semantics. Flexibility is a general property referring to the way that modelling concepts and artifacts are used for the design, development and deployment of open telecommunications services. The great variety, inherent complexity, and the increasing demand for customisation of such services raise the importance of flexibility.

Based on these assumptions, we propose a generic platform for the handling of continuous media in DCOM. This consists of primitive COM objects or services (multimedia support services), which can facilitate the construction of new telecommunications services with multimedia characteristics. More specifically, the multimedia support services, and the associated COM objects, are compatible with RM-ODP in the sense that they adopt related concepts and functionality, and thus enable a wide degree of information sharing and application interoperability. Furthermore, these services can be reused and customised, and their interfaces have been designed to allow flexibility and efficiency in achieving their implementation. This is important for telecommunications services which manipulate multimedia objects, where per-formance is critical.

While this paper deals with the flexible modelling of multimedia streams in a DCOM-based DPE, it should be noted that besides supporting modelling aspects, the control and management software of new telecommunications infrastructures needs also to support a range of QoS characteristics, the synchronisation of continuous media, and the management of underlying resources [4][16][23]. These aspects are *not* within the scope of this paper.

3 Supporting Continuous Media in DCOM

DCOM is the distributed extension to COM (Component Object Model) that builds an Object Remote Procedure Call (ORPC) layer on top of DCE RPC to support remote objects. In general, DCOM provides all the necessary facilities for the integration of heterogeneous components in a distributed environment [3][8].

However, DCOM does not satisfy the more complicated and stringent requirements of handling multimedia streams. To enable DCOM to be the basis for new telecommunications services which require the handling and control of continuous media, extra features are necessary. The most obvious requirement is that the concept of streams should be added to the DCOM object model through the introduction of stream interfaces, since at present only operational interfaces are defined.

Before focusing on DCOM, it has to be noted that COM handles multimedia information through the Microsoft DirectShow architecture (previously Microsoft ActiveMovie architecture), which incorporates the notion of streams. Apparently, the use of this notion is restricted to the environment of stand-alone multimedia capable computers with Microsoft Windows operating systems (9x, NT 4.0, 2000), i.e. DirectShow is not a distributed architecture.

For the rest of this section, in subsection 3.1 we present the key modelling abstraction, in 3.2 we present the stream communication algorithm and in 3.3 we discuss some important implementation details.

3.1 The Proposed Approach

We propose here a multimedia support platform with the introduction of a number of support services in the DCOM architecture. These services (which are used in conjunction with existing DCOM services) provide new functionality without requiring any changes to the basic underlying DCOM architectural model. The new services consist principally of two types of COM objects: devices and stream binders. These are both seen by the higher layers of (software) abstraction as normal services with standard abstract data type interfaces, but they encapsulate the control and transmission of continuous media.

COM object devices are an abstraction of physical devices, stored continuous media or software processes. They may be either sources, sinks or transformers of continuous media data (“modules”). A source is a media producer and is normally an abstraction of a media-generating hardware device, such as a camera or a micro-phone. A sink is a media consumer and is normally an abstraction of a mediarendering hardware device, such as a framebuffer / VDU or a loudspeaker. Finally, a module is both a media producer and consumer, as it accepts incoming data, processes it in some way, and produces output.

Most devices present a device dependent interface, a generic control or chain interface (IChain), and an endpoint interface (IEndpoint). The device dependent interface contains operations specific to the device modelled and is used for the management of the device. For example, a camera might have operations such as focus, pan or tilt. Furthermore, it has to be noted that all devices have to inherit from the IUnknown interface, which provides functionality required by all COM objects. This is common in most distributed object platforms e.g. in CORBA and Java Remote Method Invocation (Java-RMI).

```
interface IChain : IUnknown
{   typedef enum {in, out, inout} DeviceType;
    HRESULT GetDeviceType([out] DeviceType* DType);
    HRESULT Start();
    HRESULT StartEx([in] int NumberOfSegments);
    HRESULT Stop();
    HRESULT Suspend();
    HRESULT SuspendEx([in] int Time);
    HRESULT SuspendEx1([in] int NumberOfSegments);
    HRESULT Resume();
    HRESULT Skip([in] int NumberOfSegments);
    HRESULT GetPosition([out] int* SegmentNumber); };
```

Fig. 1. The IChain Interface

A piece of continuous media can be visualised as a chain comprising a sequence of segments or links, each of which represents an atomic unit specific to the media type in question (e.g. a frame of video) [4]. Thus, a chain is an abstraction over a continuous media source or sink that focuses on the control of the production and consumption of continuous media data. Based on this abstraction, the IChain interface provides generic operations for controlling continuous media devices and managing continuous media transmissions. It is a device- independent interface which is common to all continuous media devices. The IChain interface is summarised in Fig. 1 using (a simplified variation of) Microsoft's Interface Definition Language (M-IDL), which is an extension of the Distributed Computing Environment's (DCE) IDL.

More specifically, the GetDeviceType operation returns the type of the device under examination (producer, consumer or module), while the Start and Stop operations switch the device's information flow on and off accordingly. The functionality of the two last operations, which are the most important of the IChain interface, is based on the use of a virtual pointer (CurrentSegment) that moves through the media chain as it is played or recorded. The value of this pointer in a producer device reveals the number of segments that have been transmitted, while in a consumer device represents the number of segments that have been received. The Start and Stop operations make also use of the fact that a producer device places the outgoing segments on the OutputSegment buffer, while a consumer device places the incoming segments (before processing them) on the InputSegment buffer. It has to be noted, that the StartEx operation is a variation of the Start operation, which does not initialise the CurrentSegment pointer and produces / consumes a specific number of segments (NumberOfSegments).

```
interface IEndpoint : IUnknown
{   HRESULT GetSegment([out] BSTR* Segment);
    HRESULT PutSegment([in] BSTR* Segment);
    HRESULT SetCharacteristics ([in] long ChrSize,
                [in, size_is(ChrSize)] long* ChrArray);
    HRESULT GetCharacteristics([in, out] long* ChrSize,
                [out, size_is(ChrSize)] long* ChrArray);
};
```

Fig. 2. The IEndpoint interface

There are also operations for suspending and resuming the activity of a device (Suspend and Resume respectively). After a Suspend operation the production / consumption of segments in a device stops until the Resume operation is called or in the case of SuspendEx / SuspendEx1 for the time period specified (explicitly or implicitly) by the parameters of the operation. In both cases the value of the CurrentSegment pointer is preserved. Finally, this pointer may be located and moved using the GetPosition and Skip operations. More specifically, GetPosition returns the current value of the CurrentSegment pointer, while Skip ignores NumberOfSegment segments that were to be transferred (in the case of a producer device) or that were already transferred (in the case of a consumer device) preserving the value of the CurrentSegment pointer.

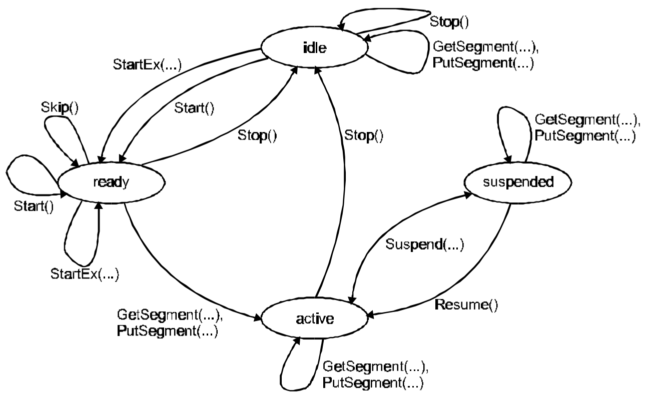

idle
Stop()
GetSegment(...), PutSegment(...)
StartEx(...)
Start()
Stop()
Stop()
Skip()
ready
Start()
StartEx(...)
GetSegment(...), PutSegment(...)
suspended
Suspend(...)
Resume()
active
GetSegment(...), PutSegment(...)
GetSegment(...), PutSegment(...)

Fig. 3. The state transition diagram of a device

Another interface which is common to all continuous media devices (device independent interface) is the IEndpoint interface. An endpoint is a connection point (a port) for a stream, and the IEndpoint interface is thus the "stream interface" of a device [15]. The IEndpoint interface abstracts over all aspects of a device which are concerned with the transport of continuous media. Essentially, as it can be seen in Fig. 2, it presents a pair of operations, GetSegment and PutSegment through which segments can be read (from the OutputSegment buffer of a producer device) or written (to the InputSegment buffer of a consumer device) respectively. With this approach, the content of a stream is not considered and it is viewed purely as a byte transport mechanism. The two other operations of the interface (SetCharacteristics and GetCharacteristics) refer to a number of transmission related characteristics used for QoS issues.

The operations inside the IChain and the IEndpoint interfaces of a specific device must take place in an acceptable and semantically correct order (e.g. for the same device a Stop operation can not be followed by a Suspend operation) to avoid unexpected results / errors. To ensure such an order, each device has a state (DeviceState), which is checked before an operation is executed. The proposed and anticipated transitions between the states of a device can be seen at the state transition diagram of Fig. 3, which also depicts the possible states of a device (idle, ready, active, and suspended).

In order to be able to control streams the binding process must be made explicit. This is done through the introduction of a binding COM object (StreamBinder). StreamBinder represents the connection between bound COM objects and, as can be seen in Fig. 4, provides an operational interface (IStreamBinder) that hides continuous media transmissions, which can be optimised by using dedicated transport protocols, entirely distinct from those used to convey control messages.

```
interface IStreamBinder : IUnknown
{ HRESULT StartSource([in] IUnknown* SourceGroup);
  HRESULT StartSink([in] IUnknown* SinkGroup);
  HRESULT Connect&Transfer([in] IUnknown* SourceGroup,
                           [in] IUnknown* SinkGroup);
  HRESULT StopSource([in] IUnknown* SourceGroup);
  HRESULT StopSink([in] IUnknown* SinkGroup);
  HRESULT SuspendSource([in] IUnknown* SourceGroup);
  HRESULT SuspendSink([in] IUnknown* SinkGroup);
  HRESULT ResumeSource([in] IUnknown* SourceGroup);
  HRESULT ResumeSink([in] IUnknown* SinkGroup);
  HRESULT DestroyConnection([in] IUnknown* SourceGroup,
                            [in] IUnknown* SinkGroup);
};
```

Fig. 4. The IStreamBinder interface

The binding action can be initiated by a COM object involved in the binding or by a completely separate object. In general, client COM objects wishing to initiate continuous media transfer, request from the StreamBinder to start the appropriate source and sink devices (StartSource, StartSink). Then, the StreamBinder establishes a stream connection between these devices and activates the transmit function (Connect&Transfer). The resulting stream can be managed by suspending (SuspendSource, SuspendSink), resuming (ResumeSource, ResumeSink), and stopping (StopSource, StopSink) the participating devices and it can be destroyed when desired (DestroyConnection). It is evident that this approach is working best in a multi-threaded DCOM environment. In such an environment, other methods can be invoked on the StreamBinder object (and thus other streams can be activated), whilst data is streamed via an existing stream connection. Additionally, this approach ensures that an operation will not be executed on a device with an incorrect, i.e. semantically unintended, role (producer / consumer). For example, the StartSource operation before calling Start on the device specified by its parameter, checks whether this device is a producer (using the GetDeviceType operation).

The StreamBinder, in the most general case, supports multiple stream connections, as it allows M sources to be connected to N sinks (without necessarily M=N), by establishing the appropriate streams between them. When it is desirable to start, stop, establish, and generally perform control operations to a number of streams simultaneously, the notion of object groups simplifies greatly the necessary code (calls to the StreamBinder operations). Additionally, it eases considerably the process of ensuring that the code reflects the correct / intended semantics, as it decreases the possibility of missing, wrong, or out of order operations on devices. This is due to the fact that errors can now appear only during the formation of object groups; an activity which corresponds to a relatively small and well structured piece of code that can easily be examined. Two typical errors that can be avoided without difficulty through the use of object groups is the execution of an operation on a device that belongs to a different stream than the one intended, and the execution of Connect&Transfer and / or DestroyConnection on two devices that (are intended to) participate on different streams.

```
interface IObjectGroup : IUnknown
{   HRESULT Join([in] IUnknown* refiid);
    HRESULT Leave();
    HRESULT Use([out] IUnknown* refiid);
    HRESULT Reset(); };
```

Fig. 5. The IObjectGroup interface

Conceptually, object groups are modelled using the COM class ObjectGroup, which collects in a group a set of related COM objects. Actually, it maintains a list of the interface references (REFIIDs) of the COM objects that belong to a specific group. The IObjectGroup interface can be seen in Fig. 5. Join and Leave operations allow new members to join the group and existing members to leave the group respectively, while Use and Reset provide access to the group current membership list.

In a typical scenario, two instances of the ObjectGroup COM class are used: a SourceGroup and a SinkGroup (which are actually the interface references of the two instances). The two lists that are maintained by these two groups, contain at corresponding positions the interface references of the sources and sinks that are going to be engaged in stream communication. Thus, the use of (the interface references of) these two groups as parameters in the operations of the IStreamBinder interface allows the invocation of (corresponding) operations on a number of COM objects (sources / sinks) at the same time. However, it must be noted that in order to increase the flexibility and support application semantics where the simultaneously establishment and control of multiple streams is not desirable, the use of object groups in the operations of the IStreamBinder interface is not mandatory. Interface references to simple COM objects (sources / sinks) can also be used as parameters.

The COM objects examined so far constitute the proposed multimedia support services for DCOM, and should be reused during the development of specific multimedia services. They may therefore have to be customised according to the specific service requirements. This activity, which is very important as it determines the practical value of the proposed approach, is supported through the use of either containment or aggregation, as DCOM allows only interface and not implementation inheritance [8].

More specifically, under containment one COM object contains another, with the outer COM object (e.g. representing a “new” enhanced multimedia device) accessing the inner COM object (representing an already used and tested “old” device) through its interfaces. On the other hand, aggregation occurs when the outer COM object exposes the interfaces of the inner COM object directly to clients. One important characteristic of this technique is that it can only be used for in-process COM object servers (i.e. DLL server modules). In this way, efficiency is improved and the handling of the tricky reference counting semantics is avoided. However, this particularity of aggregation can become a significant restriction when using the proposed API, because some of its COM objects (devices, StreamBinder) may be, as far as their clients are concerned and depending on application requirements, either local servers (implemented as EXEs) or remote servers (executed on a remote server machine). Therefore, when reusing or customising COM objects from the proposed API, the containment method is the preferred way since it enables the resulting component to operate under all possible COM server types.

3.2 The Stream Communication Algorithm

The proposed multimedia support services described in the previous section can be used for the establishment and control of stream communication in DCOM in a structured fashion. To illustrate this approach, a possible scenario is examined. According to Fig. 6, which depicts the configuration of the COM objects involved in the example scenario, two source devices (e.g. video cameras) are connected via a StreamBinder to two sink devices (e.g. VDUs), and two different streams are established between the source and sink devices.

The necessary steps that have to be followed in order to realise the two video connections (stream 1 and stream 2) between the sources and sinks of Fig. 6 using the proposed multimedia support services are the following:

Step 1 *Obtain the necessary interface references*: The interface references (REFIIDs) of the two sources (Source1UserA and Source2UserB) and the two sinks (Sink1UserC and Sink2UserC) involved in stream communication are obtained. Device dependent operations are also performed if necessary.
Step 2 *Create new instances of required services (COM objects)*: A StreamBinder instance is created and the related interface reference is obtained. Additionally (if required), two ObjectGroup instances are created and the related interface references are also obtained (SourceGroup and SinkGroup).

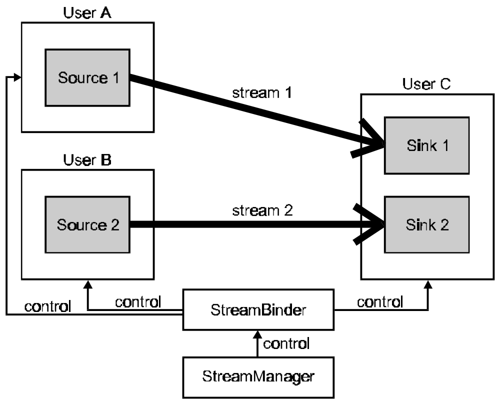


Fig. 6. An example scenario for the multimedia support services

Step 3 *Form the appropriate object groups (if required)*: Taking into account the streams that is desirable to be established (or actually considering the source and sink devices that need to be connected by streams), the REFIIDs of the sources become members of the SourceGroup [Join(Source1UserA), Join(Source2UserB)], and the REFIIDs of the sinks become members of the SinkGroup [Join(Sink1UserC), Join(Sink2UserC)].
Step 4 *Start the devices*: The sink and source devices are started [StartSink (SinkGroup), StartSource(SourceGroup)].
Step 5 *Establish connections between source and sink devices*: Associate the appropriate sources and sinks and initiate continuous media transfer between them [Connect&Transfer(SourceGroup, SinkGroup)]. Steps 4 and 5 can also take place in the opposite order.
Step 6 *Stop the devices*: When the interaction is finished the sink and source devices are stopped [StopSink(SinkGroup), StopSource(SourceGroup)].
Step 7 *Destroy connections and services*: The connections established between the appropriate sources and sinks are destroyed [DestroyConnection(SourceGroup, SinkGroup)]. Then, the StreamBinder and the ObjectGroup instances created in step 2 are also destroyed.

The above described steps constitute a kind of algorithm, i.e. a stream communication algorithm, for establishing and controlling stream connections in

DCOM. Two more steps can be added to this algorithm depending on the functionality required by some applications. More specifically, between steps 5 and 6 (i.e. while all the devices are active), the sink and source devices can be suspended [SuspendSink (SinkGroup), SuspendSource(SourceGroup)] and then, on a consecutive step, they can be resumed [ResumeSink(SinkGroup), ResumeSource(SourceGroup)].

The stream communication algorithm utilises the operations of the IStreamBinder interface to create and manage bindings between the appropriate sources and sinks. These bindings do not have to be controlled directly by the COM objects involved in the binding (i.e. the sources and the sinks), but may instead be created by third party COM objects which obtain references to interfaces owned by those COM objects. This facility eases considerably the configuration and structuring of potentially complex multimedia telecommunications services containing many per-media COM objects. A similar situation is described in Fig. 6, where the StreamManager COM object interacts with the StreamBinder and performs all the steps of the stream communication algorithm. In the general case, the StreamManager can call directly operations, both on (source / sink) devices and on the StreamBinder, and is responsible for the "encapsulation" of the (control) logic that is related with streams. To avoid errors and unexpected results, caution is needed to ensure that when an operation is executed on a device, the same or a (semantically) compatible operation is also executed on the device with which the former device is (will be) connected by a stream.

From these remarks is evident that the structure and the behaviour of the StreamManager depends on the requirements of a specific application, and on the way that this application handles streams. On the contrary, the interfaces and the functionality of the (COM objects used to model) devices and the StreamBinder are application independent and thus suitable for reuse. Actually, these interfaces (and the corresponding multimedia support services) can be considered as a high level Application Programming Interface (API) for the handling of continuous media in DCOM.

The advantages of this high level API are highlighted when taking into account that the main alternative approach for stream handling in DCOM requires the use of low level native Windows APIs (such as Win32), which is characterised quoting [20]:

- "Excessive low level details that:
  - Divert the attention of the developers from the more crucial (broader) application-related semantics and the program structure.
  - Raise the potential for errors.
  - Increase the learning effort required.
  - Hinder the development of complex applications.
- Continuous re-discovery and re-invention, in an ad hoc manner, of incompatible higher-level programming abstractions that seriously hampers programming productivity and code compatibility."

Therefore, the development of multimedia telecommunications services in DCOM benefits greatly from the use of the proposed API, because it isolates the application domain semantics from the complexities of multimedia devices and continuous media communications, by providing services based on abstract data type interfaces. Additionally, it reduces and simplifies the required programming effort, by locating all the code related with the handling of streams inside easily extensible reusable components, preventing thus developers from “reinventing the wheel” using elementary capabilities and functionality.

3.3 Important Implementation Considerations

There are a few DCOM related issues that affect considerably the implementation of the proposed multimedia support services. These issues, which will be examined briefly in this section, include class factories, access to remote COM objects, and the available threading models.

In order to be able to use a (device or a StreamBinder) COM object, an instance must be created. This is done through a special COM object called a class factory, which implements the IClassFactory interface and has knowledge on how to “manufacture” (one or more) COM objects of a particular class (this is again typical in distributed object platforms). In that way, the COM runtime gains efficiency as it is not necessary to know about all the possible object types, that might have to create, ahead of time. This functionality is based on the design pattern of a “Factory Method”, according to which, when a client wishes to instantiate a server object, a request is sent to a “Factory Object” for the corresponding class [6]. In full agreement, a class factory has to be created for every server component specified by the proposed multimedia support services. It has to be noted, that for optimisation reasons in some (not very common) cases (e.g. when a device has a large number of device specific interfaces, and depending also on their intended use), a custom implementation of the IClassFactory interface is allowed, but caution is needed to avoid possible compatibility conflicts / problems.

After performing all the necessary instantiations, a client that wishes to call operations on a (device or a StreamBinder) COM object has to obtain a pointer to a suitable interface of that object. When the desired COM object is remote, the CoCreateInstanceEx() function has to be used in a suitable manner to locate the server machine, create (an instance of) the appropriate COM object on that machine, and finally return the desired interface pointer. This function is called with an array of MULTI_QI structures as one of its parameters:

```
typedef struct _MULTI_QI {
const IID* pIID;   // pointer to an interface identifier
IUnknown * pItf;   // returned interface pointer
HRESULT hr;        // result of the operation
} MULTI_QI;
```

As it can be seen in Fig. 7, each pIID member of this array is given an IID of an interface of the remote COM object. If the CoCreateInstanceEx() succeeds, the desired interfaces can be obtained through the pointers in the pItf members. If there is an error, the hr member will receive the error code. Thus, except from the status of CoCreateInstanceEx(), the status of each element in the MULTI_QI array should also be checked, before a (valid) interface pointer can be extracted from the array.

When a client requires access to a particular remote COM object, and this object has more than one interface to which the client needs pointers, an array of MULTI_QI structures should be created, containing as many pIIDs as necessary to keep all the IIDs of the interfaces that the client will (or intends to) use on the COM object. In that way, the CoCreateInstanceEx() will be called only once and multiple calls to it, due to an incomplete (in terms of requested IIDs) MULTI_QI array, will be avoided. This tactic reduces the number of necessary RPC calls across the network, and improves the efficiency of the code especially when remote COM objects exhibit more than one interface (e.g. as in the case of COM objects used to model continuous media devices), and / or the network performance is or becomes slow.

```
// initialise the MULTI_QI structure
MULTI_QI qi[2]; // create an array of e.g. 2 structures
memset(&qi, 0, sizeof(qi)); //prepare the array for use
qi[0].pIID = &IID_IChain;     // add the 1st interface
qi[1].pIID = &IID_IEndpoint; // add the 2nd interface
// create a server COM object on the server machine
HRESULT hr=CoCreateInstanceEx(
       CLSID_CMyServer, // COM class id
       NULL,            // outer unknown
       CLSCTX_SERVER,   // server object scope
       &ServerInfo,     // name of the server machine
       2,               // length of the MULTI_QI array
       qi);             // pointer to the 1st element
                        // of this array
// check the qi codes
if (SUCCEEDED(hr))
{ // also check qi hresult
  hr=qi[0].hr; }
if (SUCCEEDED(hr))
{ // extract interface pointers from MULTI_QI structure
  m_pComServer=(ICpServer*)qi[0].pItf; };
```

Fig. 7. Using an interface of a remote server object in DCOM

Finally, shifting the focus to the internal structure of COM objects, the way that threading is performed needs to be examined. Threading involves specifying code segments that will be executed concurrently by creating, somewhere within a program, more than one thread, and ensuring the protection of shared resources, the provision of thread synchronisation, and the avoidance of deadlocks and race conditions. In DCOM, threads are established to improve performance (minimise execution time), to simplify the code, and to avoid the blocking of COM objects (e.g. to prevent the blocking of the StreamBinder when executing an operation on a device). Therefore, in the proposed multimedia support services threading is used in the implementation of the `StreamManager` and the `IChain` interface of the COM objects used to represent devices, in the interfacing with physical devices, and in the realisation of stream connections using transport protocols. When programming using DCOM, and therefore in the proposed API, except from the thread handling functions of Win32 (e.g. `CreateThread()`, `ExitThread()`, etc.), the COM threading models, i.e. the simple single threaded model, the Single Threaded Apartment (STA) model and the MultiThreaded Apartment (MTA) model can be applied [3][8].

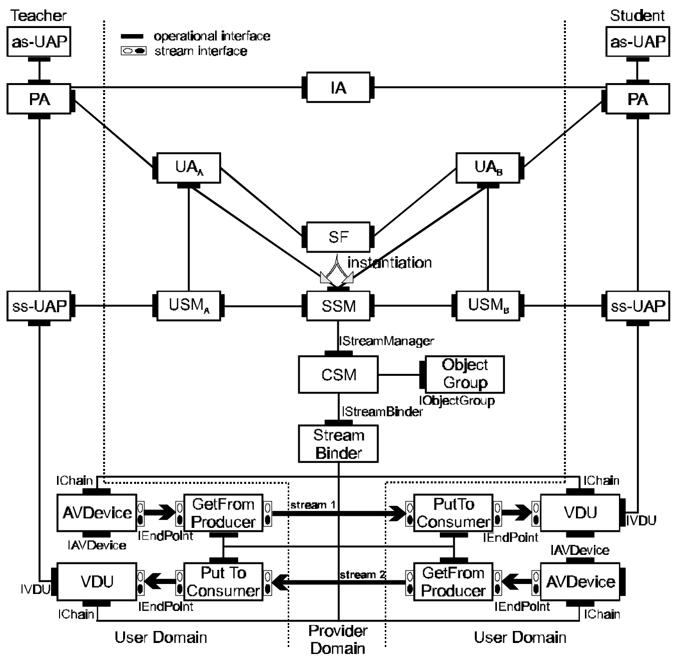
4 Validation and Experimentation

The proposed multimedia support infrastructure and the related API have been tested in several simple scenarios (like the one depicted in Fig. 6) involving different configurations of source and sink devices associated by various stream connections. It has been found that they constitute a viable, flexible, consistent, coherent and relatively intuitive way of building multimedia telecommunications services in DCOM.

To verify and reinforce these results under realistic conditions, and to determine also the true practical value and applicability of the proposed API, an extended prototype of a MultiMedia Conferencing Service for Education and Training (MMCS-ET) has been developed [18]. This service is implemented using MS Visual C++ 6.0 and DCOM on MS Windows NT 4.0, and is executed on a number of workstations connected via a 10 Mbit/s Ethernet LAN. All the interconnected work-stations belong to the same (MS Windows NT) domain and one of them functions as a primary domain controller.

The main objective of the MMCS-ET is to facilitate the establishment of an educational / training session between one teacher / trainer and a number of remote students / trainees, which is equivalent to the educational / training session that would have been established between the same people in a traditional classroom. More specifically, in a virtual classroom the teacher still has the need to manage the educational / training session. Additionally, there is also a need for audio / video (A/V) communication among all the session participants (to substitute face to face contact), text communication between only two session participants (as that achieved with the use of notepads), text communication among all the session participants (as that achieved with the use of a blackboard), file communication between session participants (e.g. for the exchange of course material), and collaboration among all the session participants in order to perform a common task. For this reason, the MMCS-ET implements a variety of scenarios supporting session management re-quirements (session establishment, modification, suspension, resumption, and shut-down), interaction requirements (audio / video, text, and file communication), and collaboration support requirements (chat facility, file exchange facility, and voting).

The computational view of the MMCS-ET in the simple case where one teacher interacts with only one student can be seen in Fig. 8. From the computational objects that appear in this figure, it is evident that the MMCS-ET is designed according to the TINA-C service architecture (version 5.0) [21]. Fig. 8 emphasises on the way that A/V communication is achieved between the teacher and the student by the establishment of two streams of opposite directions, presenting the position of the interfaces of the proposed multimedia support services for DCOM. It has to be noted that the `Communication Session Manager` (`CSM`), which is at the boundary with the resource layer, incorporates the functionality of the `StreamManager`, and that the `GetFromProducer` and `PutToConsumer` COM objects are used for the realisation of the stream connections.



UAP	*:* User Application	as-UAP	*:* access session UAP
ss-UAP	*:* service session UAP	PA	*:* Provider Agent
USM	*:* User service Session Manager	IA	*:* Initial Agent
SSM	*:* Service Session Manager	UA	*:* User Agent
CSM	*:* Communication Session Manager	SF	*:* Service Factory

Fig. 8. The main computational objects of the MMCS-ET and the position of the interfaces of the proposed API

The MMCS-ET validated the proposed API and confirmed the results of the initial tests with the simple scenarios, but also gave an insight, through several experiments, for the optimisation of the proposed API in terms of its use and its more efficient implementation in DCOM. More specifically, two types of experiments were conducted. The first type involved the application of object groups in the stream communication algorithm, examining the complexity of the resulting code (which is actually the main piece of code written by the service developer when using the proposed API) in terms of the number of necessary calls of operations to other COM objects. The number of such calls (with and without the use of object groups) for an increasing number of stream connections can be seen in Fig. 9. It has to be noted that different stream connections are established between different source and sink devices. Thus, for example, 4 stream connections imply the existence of 4 sources and 4 sinks. From Fig. 9 it is evident that the use of object groups, as the number of stream connections is increasing, reduces considerably the number of operation calls that have to be made, and therefore simplifies the code of the stream communication algorithm (together with the task of the service developer) and increases its efficiency.

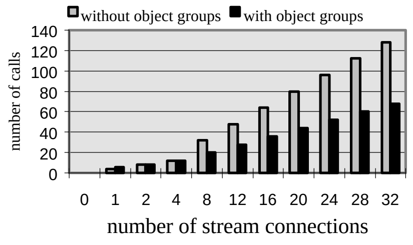


Fig. 9. Experimenting with the use (or not) of object groups in the stream communication algorithm.

The second type of experiment involved the examination of the performance (in terms of execution time) of the different COM threading models when applied to the proposed API. A choice between these models becomes especially important when the `StreamManager` creates separate threads for the instantiation of the (COM objects representing the) devices and the execution of device specific actions, for the instantiation and initialisation of the `GetFromProducer` and `PutToConsumer` COM objects, and for the instantiation of the `StreamBinder` and the execution of the stream communication algorithm. It has to be noted that only the STA and MTA models are considered because the single threaded model is really just a special type of the STA model.

Table 1. Comparison of COM threading models using the proposed API

COM Threading Models	**Time 1**	**Time 2**
STA	19.6 ms	12.6 ms
MTA	18.19 ms	11.66 ms

When each of the STA and MTA models are applied to all of the (COM objects) of the proposed API the time needed (in ms) to start (Time 1) and stop (Time 2) a stream connection between one source and one sink device is measured for each of them. Time 1 corresponds to steps 4 and 5 of the stream communication algorithm, while Time 2 corresponds to steps 6 and 7. The results of the measurements can be seen at Table 1. From this table is evident that the MTA model has a better performance (which becomes even better as the number of stream connections increases). Therefore, for the proposed API, taking also into account that there are no synchronisation issues, the MTA model is the preferred choice. Its performance superiority is mainly due to the fact that inter-thread access is direct (as all the treads are in the same apartment), requiring no proxy intervention as in the STA model.

Finally, to place the proposed API for DCOM in a more general context, and increase in that way the confidence in its use, a (high level) comparison with the approach followed by OMG for the handling of continuous media [17] is attempted, because OMG's CORBA is considered to be the main alternative to DCOM. The results of this comparison, which focuses on how continuous media communication is modelled, can be seen in Table 2.

Table 2. Comparison of modelling approaches for handling continuous media in CORBA and DCOM

Important Concepts	Modelling in OMG A/V Spec. (CORBA)	Modelling in the proposed API (DCOM)
Multimedia device	`MMDevice` interface/object	`Device` COM object
Device specific aspects	`Vdev` interface/object	`Device` dependent interface
Device control aspects	`StreamEndpoint` interface/object	`IChain` interface
Stream endpoint	`StreamEndpoint` interface/object	`IEndpoint` interface
Stream binding	`StreamCtrl` interface/object	`StreamBinder` COM object
Stream	`StreamCtrl` interface/object	`Connect&Transfer` operation (`StreamBinder` COM object)
Stream flows	`FlowConnection` interface/object	A stream has only one flow

From this table it can be easily deduced that the proposed API and the OMG A/V streams specification have the same scope as they are modelling the same basic concepts (due to the influence by the RM-ODP that have in common), albeit in different ways. Therefore, they are "conceptually compatible", although their target technological domains are divergent, facilitating thus service developers to map their designs regarding continuous media interactions easily to either a DCOM or a CORBA DPE. It has to be noted that the comparison of Table 2 wasn't extended to cover performance issues, because performance depends greatly on the actual implementation of the OMG A/V streams specification by the different ORB vendors, and because the result of such a comparison wouldn't be very important as the decision for the adoption of one of the approaches depends almost entirely on the choice of the base DPE technology e.g. DCOM or CORBA; a choice which is relatively difficult as detailed in [2].

5 Conclusions

There is a technology push in the area of multimedia communications, which is acting as a catalyst for the specification and development of new multimedia telecommunications services. These services will be deployed in a distributed object environment. Therefore, there is an increasingly important need for distributed object platforms to support continuous media interactions in a flexible manner.

In this paper, we have proposed a number of RM-ODP compliant multimedia support services together with a related API in order to enhance DCOM with continuous media support. These extensions, which offer an abstraction over stream communications and multimedia devices, do not affect the core DCOM architecture, but only add the necessary functionality in terms of additional services (DPE services).

The viability of the proposed approach was evaluated by the implementation of the MMCS-ET, which demonstrated that DCOM's features can be successfully extended to address multimedia requirements in such a way that a substantial amount of software reuse can be achieved, which is the target of flexible DPEs. It remains to be seen if DCOM will form the basis of telecommunication services in the future.

References

1. Adamopoulos, D.X., Papandreou, C.A.: Distributed Processing Support for New Telecommunications Services. Proceedings of ICT '98, Vol. III, IEEE (1998) 306-310
2. Adamopoulos, D.X., Pavlou, G., Papandreou, C.A.: Distributed Object Platforms in Telecommunications: A Comparison Between DCOM and CORBA. British Telecom. Eng. 18 (1999) 43-49
3. Brown, N., Kindel, C.: Distributed Component Object Model Protocol DCOM. Microsoft Corporation (1998)
4. Coulson, G., Blair, G.S., Davies, N., Williams, N.: Extensions to ANSA for Multimedia Computing. Computer Networks and ISDN Systems 25 (1992) 305-323
5. Dang Tran, F., Perebaskine, V., Stefani, J.B., Crawford, B., Kramer, A., Otway, D.: Binding and Streams. the ReTINA Approach. Proceedings of TINA '96 (1996)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
7. Gay, V., Leydekkers, P.: Multimedia in the ODP-RM Standard. IEEE Multimedia 4 (1997) 68-73
8. Grimes, R.: Professional DCOM Programming. Wrox Press (1997)
9. IMA: Multimedia Systems Services. IMA Recommended Practice, 2nd Draft (1994)
10. ISO: Information Processing Systems Computer Graphics and Image Processing - Presentation Environments for Multimedia Objects (PREMO), Part 3: Multimedia System Services. ISO/IEC JTCI/SC24 draft, 1996-05-15 (1996)

11. IONA Technologies: Orbix MX: A Distributed Object Framework for Telecommunication Service Development and Deployment. (1998)
12. ITU-T, ISO/IEC Recommendation X.902, International Standard 10746-2: ODP Reference Model: Descriptive Model. (1995)
13. Kinane, B., Muldowney, D.: Distributed Broadband Multimedia Systems Using CORBA. Computer Communications 19 (1996) 13-21
14. Li, G.: DIMMA Nucleus Design. Technical Report, APM Ltd., Cambridge, UK (1997)
15. Martinez, J.M., Bescós, J., Cisneros, G.: Developing Multimedia Applications: System Modelling and Implementation. Multimedia Tools and Applications 6 (1998) 239-262
16. Mühlhäuser, M., Gecsei, J.: Services, Frameworks, and Paradigms for Distributed Multimedia Applications. IEEE Multimedia (1996) 48-61
17. Object Management Group: Control and Management of Audio / Video Streams. OMG document telecom/98-06-05 (1998)
18. Papandreou, C.A., Adamopoulos, D.X.: Design of an Interactive Teletraining System. British Telecom. Eng. 17 (1998) 175-181
19. Schmidt, D.C., Levine, D., Mungee, S.: The Design of the TAO Real-Time Object Request Broker. Computer Communications 21 (1998) 294-324
20. Schmidt, D.C., Cleeland, C.: Applying Patterns to Develop Extensible ORB Middleware. IEEE Com. Mag. 37 (1999) 54-63
21. TINA-C: Definition of Service Architecture. Version 5.0 (1997)
22. TINA-C: Engineering Modelling Concepts - DPE Architecture. Version 2.0 (1994)
23. Waddington, D., Coulson, G.: A Distributed Multimedia Component Architecture. Proceedings of EDOC ’97 (1997)