# Distributed Object-Based Continuous Media Applications*

Carlos A. G. Ferraz
Departamento de Informática
Universidade Federal de Pernambuco
Caixa Postal 7851
50732-970 Recife, PE, Brazil
E-mail: cagf@di.ufpe.br

**Resumo**

Este artigo descreve uma abordagem para o desenvolvimento de aplicações de mídias contínuas baseadas em objetos e distribuídas, considerando o suporte a mídias contínuas e reuso de software. A abordagem é baseada no modelo ODP, que é baseado em objetos. Os pontos de maior interesse envolvidos no projeto e construção de aplicações de mídias contínuas distribuídas são a modelagem de mídias contínuas, o controle de taxas e a transmissão de mídias deste tipo. O modelo de estruturação das aplicações é discutido, e alguns resultados do desempenho de uma aplicação desenvolvida usando a abordagem proposta são apresentados, mostrando que o modelo é adequado.

**Abstract**

This article describes an approach for the development of distributed object-based continuous media applications, considering support for continuous media and software reuse. The approach is based on the model for open distributed processing (ODP), which is object-based. The major issues involved in the design and construction of distributed continuous media applications are the modelling of continuous media, rate control and continuous media transmission. The model for structuring the applications is discussed, and some performance results of an application developed using the proposed approach are presented, showing the adequacy of it.

## 1 Introduction

Ferrari *et al* [7] define *continuous media* as

"... to mean digital data that is generated/consumed isochronously at some granularity (e.g., motion video displayed at 30 frames per second)."

*This work was financially supported by CAPES, Brazil. Grant no. 2577/91-8.

The term 'continuous media' is particularly associated with video and audio.

This article discusses the development of distributed continuous media applications based on objects, including the support for continuous media and software reuse. The motivation is that the use of continuos media in various applications (e.g. teleconferencing) has increased at a high rate, and methodologies and technologies need to be developed for the construction and support of such applications, which are usually distributed.

According to Steinmetz in [23],

> "a multimedia system is characterized by the integrated computer-controlled generation, manipulation, presentation, storage, and communication of independent discrete and continuous media."

The nature of multimedia applications implies that several data streams have to be handled in parallel [21].

A distributed approach to building continuous-media applications is important to deal with aspects such as

- storage of information: the large quantities of data involved in audio or video applications demand efficient use of storage resources, often distributed across networks. Thus, applications need to be capable of using data which are physically distributed;
- performance: the complex and possibly concurrent activities existing in the applications require a great deal of processing power which can be more efficiently achieved through distributed processing, using multiple processors distributed over networks;
- multiple use: different users may share applications, possibly working collaboratively, and applications may share services. Thus, applications should be structured to use modules designed for general purposes or multiple use.

Concurrent activities sometimes need to synchronize, as for example in the presentation of annotations and continuous documents. Synchronization of continuous media (e.g. voice and video) is not an easy task, especially when user manipulation of the presentation is allowed. The design of continuous media applications has to take into account the particular characteristics of the media involved, quality of service requirements and interaction forms.

Support for application distribution has been developed, particularly in the form of software platforms sitting in between operating systems and the application level. Moreover, developments in operating systems and hardware to support distribution and continuous media encourage the development of distributed continuous-media applications.

## 2 Open Distributed Processing

System models and programming support are necessary for the development of distributed applications. Facilities have been developed to allow the construction of such applications, i.e. technologies including tools and software platforms to support client/server applications. System-dependent applications are restricted by the number of platforms on which they can run or with which they can exchange information. A solution to this is seen in *open platforms* for the programming of applications. Open platforms hide the differences

between underlying systems and make application development easier. The applications no longer need to call device- or system-dependent services.

There has been research in order to establish means for *open distributed processing*, i.e. to allow systems or applications to be distributed and to execute

- on machines from different vendors, and
- independently of operating systems.

The basic needs for distributed applications are support for communication, concurrency and synchronization. Applications can be built on top of distribution platforms, which provide facilities – such as RPC for interprocess communication, *threads* for concurrency, and *event counters* and *sequencers* for synchronization (see section 3) – capable of decoupling application developers from operating system and network primitives. If some facilities, such as thread support, are provided by the operating system, the platform makes use of them; if not, they are built within the platform implementation. In either case, the platform hides the detailed mechanisms from the applications [15].

Other important issues in the development of distributed applications are how the components of an application know about each other's existence and the specification of components' binding. Elements like the OMG[1]/ORB (Object Request Broker) insulate clients from the mechanisms used to communicate with, activate, or store server *objects* [19]. Applications can be built as a collection of server objects that specify *interfaces* for communication and offer them to a *broker* or *trader*, and client objects that search in the broker for interfaces (by type) that are said to provide the services they want to use. These technologies are the basis of our work, and as such, are further discussed in this article.

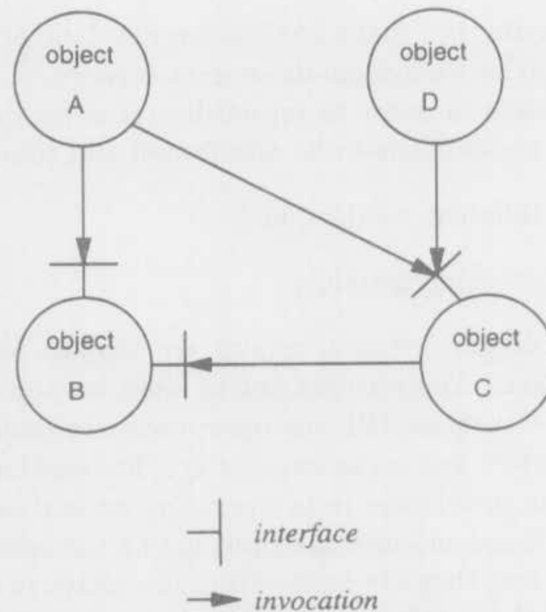
## 2.1 Object-Based Approach

Object-oriented (or object-based) technology comprises methods, tools, and frameworks used to build software systems from objects; object orientation is also a possible way to achieve such goals as extensibility and reusability.

Modelling a distributed system as a distributed collection of objects, that have state and behavior and offer interfaces for interaction, appears both natural and appropriate. In Figure 1, object B provides two possibly different interfaces and serves objects A and C; object C is a client of B and offers an interface that provides service shared by objects A and D; A and D are clients only.
The following concepts, identified and discussed in [22], confirm the appropriateness of objects for use in distributed systems:

- All objects embody an abstraction.
- Objects provide services.
- Clients issue requests.
- Objects are encapsulated.
- Requests identify operations.

[1]Object Management Group.




Objects interact by invoking *operations*
grouped in interfaces provided by each other.

Figure 1: Object model.

- Requests can identify objects.
- New objects can be created.
- Operations can be generic.
- Objects can be classified in terms of their services.
- Objects can have a common implementation.
- Objects can share partial implementations.

Object orientation has been explored in the development and use of distributed operating systems (e.g. Chorus [20]), intermediate architectures (e.g. CORBA [18, 19] and ANSA [2]) and applications and environments, such as PREMO [10, 24], a presentation environment for multimedia objects under development by ISO/IEC JTC1/SC24/WG6 [11].

## 2.2 Standardization Initiatives

Two major, cooperating, initiatives for the standardization of open distributed processing, one by the International Standards Organization (ISO) and the other by the Object Management Group (OMG), an industry consortium, are presented in the following.

### 2.2.1 ISO/ODP

The ISO's Open Distributed Processing Reference Model (ODP-RM) [12] covers the many aspects of the operation of a distributed system. The ODP architecture provides means

for consistency checking in the relationships between the human interface, the programming interface and the OSI protocols. Within the architecture, support of distribution, interworking, interoperability and portability can be integrated.

According to the ODP model, a system is considered from different *viewpoints*, which reflect specific design concerns [13, 14]. There are five viewpoints considered, and these are briefly defined as follows:

(a) Enterprise viewpoint: this is concerned with the business and management policies and human (user) roles with respect to the systems and their environment;

(b) Information viewpoint: this is concerned with the description of information models, i.e. information sources and sinks and the information flows between them;

(c) Computational viewpoint: here the concern is with the algorithms and data structures of the distributed system;

(d) Engineering viewpoint: in this viewpoint there is concern with the mechanisms and transparencies that support distribution;

(e) Technology viewpoint: this is concerned with the components and links from which the distributed system is constructed.

While the five viewpoints are relevant to the design of distributed systems, the computational and engineering ones are more specifically associated with the use and construction of these systems.

### 2.2.2 OMG/CORBA

One of the objectives of the OMG is the definition of an architecture for distributed applications using object-oriented techniques. The architecture consists of four elements:

(a) the Object Request Broker (ORB), which is the object interconnection bus, or the communications element, for handling the distribution of messages between application objects;

(b) the Object Services, which extend the capabilities of the ORB, allowing the logical modelling and physical storage of objects – the services include naming, persistence, life-cycle management and concurrency control;

(c) the Common Facilities, divided into two categories, called *horizontal* and *vertical*. The horizontal common facilities provide information management, systems management, task management and user interface services. The vertical common facilities are to be provided in many application domains (e.g. health and finance) through class interfaces;

(d) the Application Objects, which are specific to end-user applications, built on top of the ORB, the object services and common facilities.

The Common Object Request Broker Architecture (CORBA) initially described the interface technology for the ORB portion of the reference model. CORBA 1.1 [18] defined the Interface Definition Language (IDL) and the application programming interfaces (APIs) to enable interaction between client- and server-objects within a specific implementation of an ORB. CORBA 2.0 deals with the interoperation of ORBs from different vendors [19].

# 3 ANSAware

ANSAware is an implementation of ANSA (Advanced Networked Systems Architecture), an architecture for open distributed processing, that conforms with the ODP model. which supports the design and construction of distributed applications, and is not constrained by network structure, or heterogeneous hardware and operating systems [2]. The main features of ANSAware are described as follows.

**Services.** The basic building block of ANSA is a *service*; and a service is provided at an *interface*. A component or object purely described in terms of the way it provides or uses services is called a *computational object*. A computational object may have several interfaces, each offering the same or different services (c.f. Figure 1) - a service is a collection of *operations*. Each instance of a service interface has a unique identifier called an *interface reference*.

Services are divided into *application services*, which are specific to the task to be performed by the system or application, and *architectural services*, which provide functions for naming and finding services, access control and management within a distributed system.

A **trader** registers service offers made by service providers and returns information about the available services accessible to clients that request them. When a client requests a particular service, the trader matches the request with existing offers by using interface types, context names and service properties in combination as selection criteria. This is how the separate parts of a distributed application can find each other on demand. The control of services available on a network is completed by two additional service providers. **node managers** and **factories**. Trader, node managers and factories cooperate to allow. for example, dynamic instantiation of objects [3]. (There is also dynamic instantiation of interfaces, discussed later.)

**Transparencies.** ANSAware allows interaction between objects without needing them to know each other's physical location, and allows a uniform style of interaction irrespective of the objects' construction or environment, or whether they are local or remote. These are called *location transparency* and *access transparency*, respectively.

**Languages.** A computational object is described using the following two languages:

- **IDL.** This definition language is used to specify interfaces, e.g.

```
ExampleIntf : INTERFACE =
-- interface name

NEEDS CommonIntf;
-- specification inheritance

BEGIN
-- ============================================================
-- type definition

    Status: TYPE = {Succeeded, Failed}

-- ============================================================
-- operation signatures

-- the following specification enables a synchronous call
    Register: OPERATION [ offered_service: ansa_InterfaceRef]
```

```
    --          interface references can be passed as arguments
                RETURNS [ status: Status;
                          handle: CARDINAL;
                          req_service: ansa_InterfaceRef ];
    --                ... and received as results

    -- the following configures an asynchronous call
    --          in which results are not needed
       Dereg: OPERATION [ handle: CARDINAL ]
              RETURNS [ ];

    END.
```

- **PREPC.** This language provides a means for embedding invocations of interface operations in C source files. The generic syntax for an operation invocation is:

```
! { results } <- interface_ref$operation (args) exceptions
```

**Capsules.** Computational objects are potentially remote from one another. When they are compiled (and are then called *engineering objects*), operation invocations are translated into calls to the local *nucleus*, which manages the resources of a *node* and assigns them to engineering objects called *capsules*. A capsule is the unit of autonomous operation within ANSAware.

If ANSAware is supported by a multi-tasking operating system such as UNIX, then one node may support various capsules, and a capsule, in this case, is a UNIX process. Capsules have the following capabilities:

- encapsulation, i.e. a capsule is a protection domain and an atomic unit of failure;
- provision for concurrent activities, and synchronization and ordering of such activities within each capsule;
- communication with other capsules;
- preservation of state between interactions, unless a failure occurs;
- provision for creating other capsules.

**Concurrency.** ANSAware provides concurrency in a multi-threaded environment. A *thread* is an independent execution path which can be executed concurrently with other threads. The resources a thread requires (a stack to store its local variables and function return links, and a register dump area) are provided by a virtual processor called a *task*. Tasks are the units of actual concurrency provided by the system, while threads are the units of potential concurrency – tasks are more expensive than threads in terms of memory. A thread has to be assigned to a task, and so tasks can service a queue of threads [4]. Components have multiple threads and may optionally support multi-tasking.

ANSAware allows concurrent activities within capsules, and thus, provides an inter-task synchronization mechanism that permits objects to control the ordering of events directly. The mechanism consists of *eventcounts* and *sequencers*:

- an eventcount is responsible for counting the number of events of a given type that have occurred so far. It is associated with
  - an *advance* primitive, which increases the value of the eventcount by 1, indicating the occurrence of an associated event,

- a *read* primitive, which reads the value of the eventcount, and
- an *await* primitive, which blocks the caller until the eventcount is equal to or exceeds the given value;

sequencers can be used to help ordering events, because eventcounts alone cannot do that. A sequencer is associated with a *ticket* operation, which returns the current value of a sequencer and atomically increments the sequencer's value.

**Communication.** Communication between capsules is done via the services they may provide to each other. They communicate by invoking operations (via RPC) at service interfaces whose references they know – one well-known interface reference is the reference to the trading service. Interface instances can be created and destroyed dynamically, and after creation their references can be *exported* to the **trader** so that they can be *imported* by clients. Any client which possesses an interface reference can use the service provided at that interface. In an application, interface references can be passed as arguments and returned as results of operations. In this case, such references are not known outside the application. In any application, at least one interface reference generally has to be exported. Figure 2 shows the use of trader's service and non-traded interfaces (the example interface (`ExampleIntf`) given before is considered as **s1**). The following steps, shown in

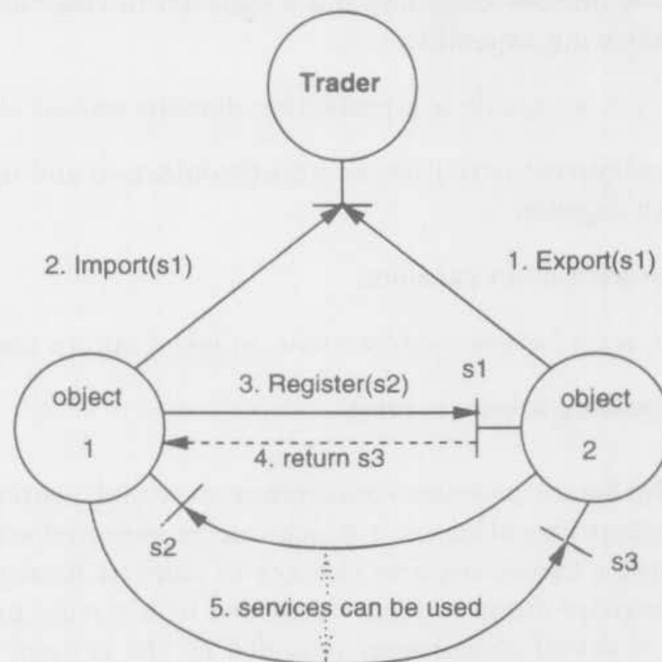



Figure 2: Traded and non-traded services.

the figure, are explained:

1. object 2 creates an instance of the interface **s1** and exports the interface reference to the Trader, offering the service provided via **s1**;
2. object 1 asks the Trader to return the reference of the interface type **s1**;
3. in possession of the reference of the interface **s1**, object 1 creates an instance of the interface **s2** and calls an operation in **s1** to register its reference, offering the service provided at **s2** to the object that provides **s1** (i.e. object 2);

4. as a result of the operation Register, object 1 obtains the reference of the interface s3;

5. the services provided at the interfaces s2 and s3 are used by their respective client-objects.

In the communication between a client and a server, transparency in the remote procedure calls is achieved through routines called *stubs*. Interface specifications are input to a stub generator, which produces both the client stub and the server stub, and these are then put into the appropriate libraries. When the client is compiled, the client stubs are linked into its binary, and the same occurs with respect to the server; i.e. the server stubs are linked with it when it is compiled [25, *pp. 420–427*].

# 4 The Approach of this Work

The approach used in the work was developed through the design and construction of an application, called The Annotation of Continuous Media [8], which combines voice with the presentation of continuous media documents in the form of music, video clips, etc., allowing such documents to be voice-annotated. Annotations are remarks that refer to specific points or segments of a document. Since the application deals with time-dependent media (voice, sound, video, etc.), the reference chosen to link annotations and the respective document-segments is *time*, enabling synchronization between them. Figure 3 shows an example of an annotation that refers to a scene of a video clip. As the annotation refers to scene 2, the annotation and this scene should be synchronized, giving significance to the contents of the annotation.

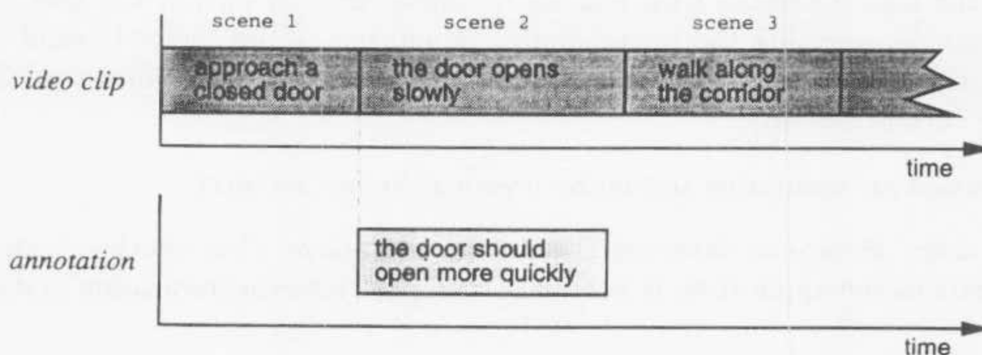



Figure 3: Example of an annotation.

Given the continuous nature of the observation, annotations are made relative to the presentation time. The application is able to provide instant access to annotations and locations in the continuous documents through a graphical user interface (GUI) including a timeline – see Figure 4.

In the GUI, three regions are distinguished: the timeline, the annotations display, and the buttons area. The slider in the timeline represents the time passing and can be dragged, forwards and backwards, for time manipulation by the user. The annotations display can show rectangles, representing existing annotations, and a line, representing an annotation that is being recorded; by clicking on a rectangle, the user can select the corresponding annotation to be played – notice that in this case time is manipulated, and

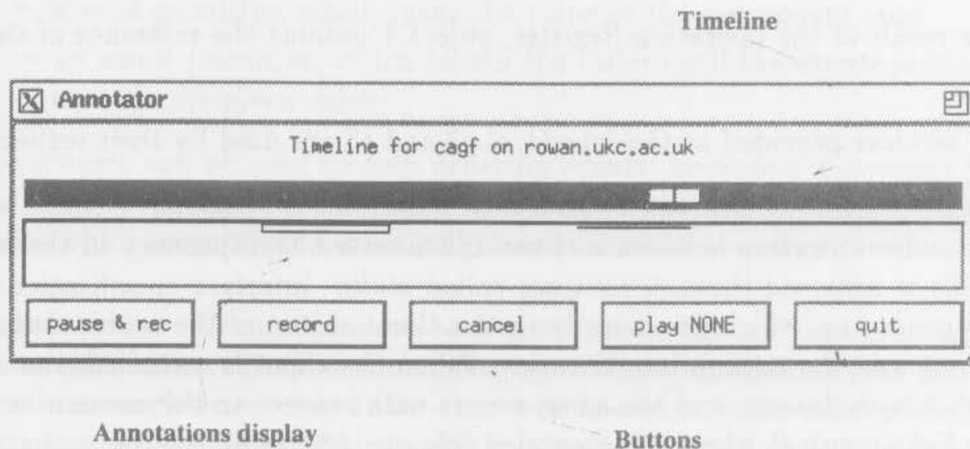



Figure 4: The GUI of the Annotator.

any change of time affects both the presentation of the (continuous media) document that is being observed and of annotations. The buttons in the GUI allow the user to execute other actions:

- to pause the presentation of the observed document and record a comment (annotate);
- to annotate simultaneously with the presentation;
- to cancel an annotation;
- to choose whether an annotation can be played or not. It can be played either as the logical passing time reaches its initial time or only if the user clicks on a rectangle, selecting the corresponding annotation – the choice is valid for all the existing annotations, and the user may change mind as many times as wished during the current session;
- to finish an annotation session by pressing the button `quit`.

It is clear, therefore, based on the description above, that synchronization among components of the application is necessary. Support for synchronization is discussed as follows.

## 4.1 Basic Components

The basic objects of the model for structuring continuous media applications can be seen in Figure 5.

The model identifies a class of reusable continuous media subsystems which can be integrated into applications via their control interfaces, but which encapsulate the details of continuous media transmission and associated resource management. Thus a continuous medium subsystem consisting of a rope server, a storage server and a server for presentation/capture of the continuous medium can be encapsulated so that its only external interactions are via the control interfaces which start and stop activities, report events and control the rate of the presentation. The details of the medium handling are within the encapsulation boundary of the subsystem.

Figure 5: Structuring continuous media applications in terms of support for synchronization and streams.

In the model, there is thus a clear distinction between *rate control* and *stream support* – achieved by identifying a *stream transmission subsystem* involving the continuous medium storage server and the device server. Rate control is independent of the stream transmission subsystem and is jointly performed by the server that manages the *rope* abstraction, based on the concept of *voice rope* introduced in [26] as a sequence of segments containing continuous media data, its clients in the application, and the storage server. The separation from continuous medium transmission makes the rate-control mechanism reusable in a number of different applications. Continuous medium transmission is encapsulated so that optimised stream handling is carried out by the storage server and the server that controls the devices for continuous medium presentation/capture. It is this server that does most of the buffering control to support the real-time requirements of a continuous medium.

### 4.1.1 Stream handling

Transmission of continuous media data from source to sink in a distributed environment has been considered in several works [17, 1, 9, 16, 6] . Some of the issues involved are definition of sample size, communication delays, transfer mechanism (RPC, etc.) and buffer management.

In the application discussed in [8], which involves the annotation of music, audio transfer occurs between the storage server and the audio server (providing the *stream support* in figure 5) and this is done by repeated RPC to provide flow control. The operation used for data transfer (**Spurt**) can receive 512 bytes of audio data, which are put in the input queue for the device driver, and can return another 512 bytes of audio, taken from the output queue (see figure 6).

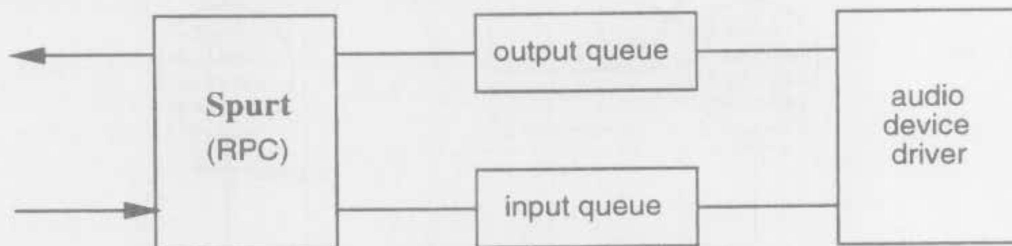



Figure 6: Buffering in the audio server.

The balance moves between the input and the output queues depending on the average speed of the RPC and presentation mechanisms. In non real-time operating systems, such as UNIX, if data buffering is limited, audio can be disrupted by delay caused by privileged activities. If there is overloading there can be data loss. Therefore, the audio server includes a control mechanism to stabilise the loop buffering. It is the audio server which is responsible for maintaining the rate of presentation through control of buffering. Observe that this control can be concentrated at the audio server, with buffering at the storage server being sufficient to support disc operations (read/write). These disc operations do not need to be positioned accurately in real time, but only need to meet the rate constraints, and so there is no need for an additional component to control rates in the transmission layer. Applications are freed from the task of doing additional buffering control, since it is done by the device control server (e.g. audio server) and the storage server used by the application.

## 4.2 Main Ingredients of the Approach

The major issues involved in the design and construction of distributed continuous media applications are:

- how continuous media are modelled: the way continuous media are structured and represented is important to define how they can be manipulated;
- rate control: this relates to stream self- and mutual-synchronization, i.e. maintaining presentation rates and synchronization of multiple streams; and
- continuous media transmission: this relates to resource control aspects of stream handling, or how the real-time requirements of continuous media presentation can still be satisfied, considering the variation of communication delay.

The following ingredients of the model proposed in this work provide solutions to the problems described above:

- the rope server: because it lets its clients abstract from the complexity of manipulating continuous media; in particular, storage and transmission – the *rope* abstraction allows the clients to see it as a simple structure, requesting via a simple interface for a rope to be built, edited, stored, played, recorded, etc., without knowing what type of medium it represents and how the medium is structured; similar rope structures can be used as a basis for video or audio applications;
- the rate-control mechanism: because it allows distributed application objects to perform time-dependent activities efficiently according to logical clocks that can themselves be periodically synchronized in a *rate community*, via a well-defined and clear interface, called `rate`. A *rate community* is formed by objects interested in synchronizing some of their activities. Each object defines logical clocks with respect to real time. For each set of activities to be synchronized, a rate community is established orchestrating local clocks to give global synchronization; thus, objects can be engaged in different communities if they wish to have some of their concurrent activities controlled by specific logical clocks (see Figure 7) – internally, synchronization can be achieved through the combination between the rate-control mechanism and event counters and sequencers.

  Each object provides interfaces of the type rate in order to keep the clocks synchronized in terms of *position* (in time), *speed* and *direction*. Each clock is then used within an object to ensure that the associated fine-grain activities are performed on time. The rate-control mechanism can be applied to many time-varying processes, not specifically related to continuous media (e.g. those providing graphical feedback of progress), making it highly reusable.
- the continuous medium subsystem: in particular, the mechanism implemented by the continuous medium device server makes the *stream transmission subsystem* (involving the storage server and the device server – see *stream support* in Figure 5) incorporate all the real-time links to the supporting system – they form part of the interface to the device – making the control interfaces as simple as possible; the minimum application involvement is ensured by the encapsulation of stream handling in the reusable components of the subsystem.

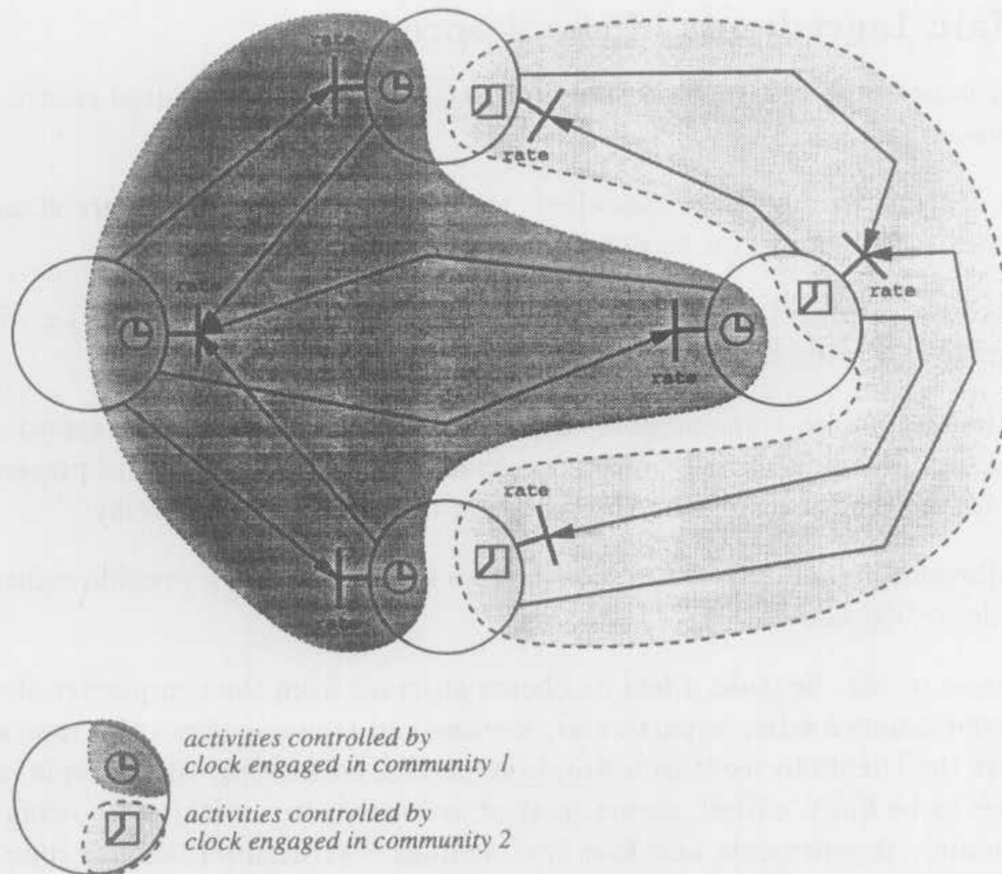



Figure 7: Rate communities in an application.

Table 1 shows a summary of the ingredients, plus the advantages and disadvantages of the solutions provided.

## 4.3 Related Work

The approach taken by Coulson *et al.* [6], for example, identifies

1. *explicit representation of continuous flow* in the computational viewpoint, and *continuous commitment/ resource reservation* in the engineering viewpoint, with regard to continuous media support, and
2. *programming specification* and *synchronization in the communication subsystem plus operating system support*, in the respective viewpoints, with regard to real-time synchronization.

The first of these is a common requirement for any continuous media system, but the degree to which an application developer needs to be involved with engineering detail can be minimized by careful choice of system components and their interfaces.

With respect to the second point, the approach describes things in terms of close links between communication and synchronization, whereas our approach has weak links, via well-known components. In our case, the separation between synchronization and communication lets the choice of rate control be separated from continuous media transmission, making the synchronization out-of-band; continuous medium transmission is encapsulated

| Issue | Ingredient | Solutions Provided | Advantages | Disadvantages |
|---|---|---|---|---|
| Continuous media modelling | Rope server | Rope abstraction and simple interface | • Reusability<br>• Client objects abstracted from details of continuous media manipulation<br>• Continuous media seen as a simple structure | Simple abstractions are fine, as long as they fit in with the application, but if one needs to do something that is not modelled (e.g. speech recognition) then a more detailed control may be needed |
| Stream synchroni-sation | Rate control mechanism | Logical clock, rate community, well-defined interface and clear operations | • Wide applicability<br>• Easy manipulation | Possibly less precise than special-purpose synchronization mechanisms |
| Continuous media transmission | Stream sync. subsystem | Encapsulated stream handling | • Very simple real-time links between the sub-system and the supporting sys.<br>• Minimum application involvement<br>• Reusability<br>• Portability | Possibly less efficient than system-supported mechanisms, because there can be extra latency following control operations, and some additional communication cost |

Table 1: Main model ingredients to be included in distributed continuous media applications.

in optimised stream handling carried out by a stream transmission subsystem, which is integrated into the application (as shown in Figure 5).

Observe that the features proposed by Coulson and colleagues are to be included in the underlying systems, whereas our approach proposes that reusable components are integrated into the application, so that the application developer does not need to know about details of the continuous media support and works at an appropriate level of abstraction. In fact, both approaches aim at reducing the degree of visibility of the underlying mechanisms by the applications - the stronger the abstraction, the better. The main differences are at what level the mechanisms are provided - Coulson's approach is to provide them in the underlying systems - and in the demands imposed on a platform by a continuous media application. Our system is, in this sense, easier to port to another similar, object-based platform, since it does not require any special service to be included in the platform or special support from the communication subsystem or from the operating system - the mechanisms are encapsulated in it, allowing it to provide adequate and efficient support for the real-time requirements of continuous media in open distributed processing, and making the applications integrated with it highly portable.

## 4.4 Building Larger Systems

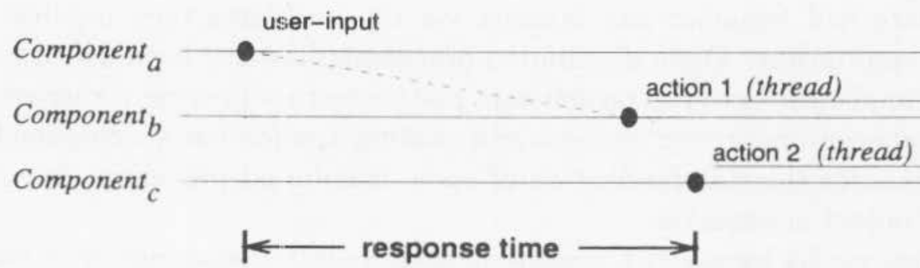
The fact that the separation of transport and synchronization reduces the number of constraints to be met simultaneously when selecting protocols and software components is fundamental to the construction of larger systems - one can have different transport in different parts of the system, while retaining any necessary synchronization. Another important point is that a rate community can be decomposed, forming a *rate hierarchy*, with the server in a given community being a client of the rate server in another community, allowing the communities, and therefore, their members, to synchronize - in a community of communities. This gives better scaling properties than an n-way interaction depending on a specific transport protocol. It also fits better with software structure, because a subsystem can have its own rate community, which is synchronized with the application as a whole, whereas the low level approach leads to problems of deciding which level of software owns the synchronization hooks.

# 5 Measurements

Response times to user requests were measured in the application developed. In this work, *response time* was defined as the time-interval between the user's input and the beginning of the last action to be taken to respond to the user-request - see illustration in Figure 8.

The maximum acceptable response time in the application was defined as 80 milliseconds, considering studies discussed in [5, 17, 23]. The times measured were related to the following events:

- *button click*: it was measured the time when the user clicked the button to start an operation;
- *slider movement*: the time when the slider started or finished moving in the timeline, according to the operation requested, was also considered; and




*Component$_a$* controls the GUI that accepts user inputs.

Figure 8: Definition of response time.

- *presentation*: the time of the start or stop – depending on the operation requested – of the continuous medium presentation was also measured.

The response times corresponded to the operations `play`, `pause`, `continue`, `skip` (`forward` and `backward`), intended to affect the presentation of the continuous medium document, and the average results obtained are given in Figure 9.

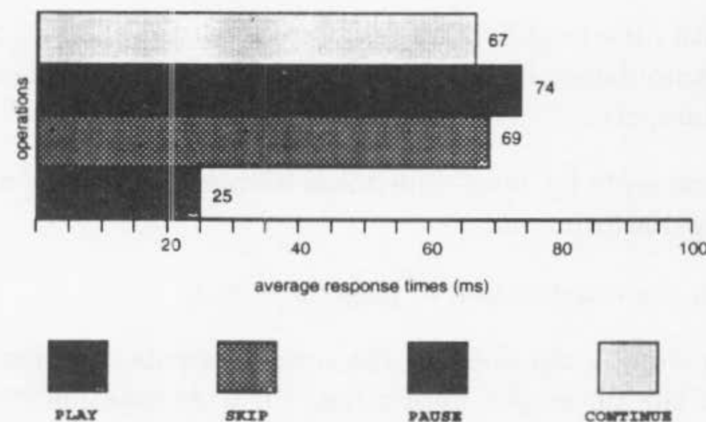



Figure 9: Average results of the response times measured in the application.

Excessive communication, granularity of data transfer (to/from disk and over networks), buffering control, and trade-offs imposed by the applications are some factors that can affect the performance of distributed applications. Additionally, machine loading, network traffic, etc., are factors that can result in differences in performance. (Different hosts were used in the different testing-sessions trying to minimise the influence of these additional factors in the results.)

The factors above were considered and improvements were achieved, in relation to the selected operations, mainly through control over communication and removal of unnecessary delays. The different performance results shown in Figure 9 can be explained by the differences between the algorithms of the operations (see [8]).

# 6 Conclusions

This article has discussed the development of distributed continuous media applications. It has been shown that modelling a distributed system as a collection of distributed objects,

that have state and behavior and interact via the interfaces they provide, seems both natural and appropriate. Open distributed processing has also been discussed as to allow systems and applications to run on software platforms that provide transparencies, hiding differences between underlying systems and making application development easier. The main initiatives for the standardization of open distributed processing have adopted the paradigm of object orientation.

Continuous media have strict synchronization requirements, either in terms of intra-stream synchronization (i.e. maintenance of presentation rates) or in terms of inter-stream synchronization (i.e. synchronization of multiple streams). Other items, like, for example, storage, intrinsically require continuous media applications to be structured for distribution. The applications can be built on top of distribution platforms, which provide facilities for communication, concurrency and synchronization.

Data modelling, rate control and transmission are the major issues identified in the design and construction of distributed continuous media applications. The proposed approach

- models continuous media generally,
- provides a high level of abstraction through simple and well-defined interfaces,
- is reusable, with little modification, in a number of applications that require synchronized time-manipulation in the presentation of continuous media, including audio, video, animation, etc.,
- satisfies requirements for open-endedness without asking for additional support in the underlying platform, and
- can be used in the construction of larger systems.

The satisfactory results obtained in the measurements of response times related to some operations in the developed application (The Annotation of Continuous Media) show the efficiency of the synchronization mechanism and the adequacy of the structuring model. The studies will continue with the development of larger applications and also considering the possibility of multiple use, especially for synchronous collaborative work, so that the synchronization mechanism can be exploited further. The applications should also be developed on other platforms that use a similar object model.

# Acknowledgements

I thank Prof. Peter Linington, my supervisor during my PhD (1991-95) at the University of Kent at Canterbury (UKC), England, who contributed with his valuable opinion and guidance in my work. I also thank members of the Distributed Systems and Networks group at UKC, in particular David Barnes and Li Ning.

# References

[1] D. Anderson and G. Homsy. "A Continuous Media I/O Server and its Synchronization Mechanism". *IEEE Computer*, 24(10):51–57, 1991.

[2] APM. *An Overview of ANSAware 4.1.* Architecture Projects Management Ltd., Cambridge, UK, 1993.

[3] APM. *ANSAware 4.1: Application Programming in ANSAware.* Architecture Projects Management Ltd., Cambridge, UK, 1993.

[4] APM. *ANSAware 4.1: System Programming in ANSAware.* Architecture Projects Management Ltd., Cambridge, UK, 1993.

[5] R. Coats and I. Vlaeminke. *Man-Computer Interfaces: An Introduction to Software Design and Implementation.* Blackwell Scientific Publications, 1987.

[6] G. Coulson, G. Blair, J. Stefani, F. Horn, and L. Hazard. "Supporting the Real-Time Requirements of Continuous Media in Open Distributed Processing". *Computer Networks and ISDN Systems*, 27(7):1231–1246, July 1995.

[7] D. Ferrari, A. Grupta, M. Moran, and B. Wolfinger. "A Continous Media Communication Service and its Implementation". In *Proceedings of GLOBECOM'92*, pages 220–224, Orlando, Florida, 1992.

[8] C. Ferraz. *The Annotation of Continuous Media.* PhD thesis, Computing Laboratory, University of Kent, Canterbury, England, 1995.

[9] D. Gemmell, H. Vin, D. Kandlur, P. Rangan, and L. Rowe. "Multimedia Storage Servers: A Tutorial". *IEEE Computer*, 28(5):40–49, May 1995.

[10] I. Herman, G. Carson, J. Davy, D. Duce, P. ten Hagen, W. Hewitt, K. Kansy, B. Lurvey, R. Puk, G. Reynolds, and H. Stenzel. "PREMO: An ISO Standard for a Presentation Environment for Multimedia Objects". In *ACM Multimedia'94 Conference*, 1994. (8 pages).

[11] ISO. *Information Processing Systems - Computer Graphics and Image Processing - Presentation Environments for Multimedia Objects (PREMO).* International Standards Organization, 1995. Committee Draft ISO/IEC 14478.

[12] ISO. *Open Distributed Processing - Reference Model.* International Standards Organization, 1995. ISO/IEC 10746.

[13] P. Linington. Introduction to the Open Distributed Processing Basic Reference Model. In J. de Meer, V. Heymer, and R. Roth, editors, *Open Distributed Processing*, pages 3–13. Elsevier, 1992.

[14] P. Linington. "RM-ODP: The Architecture". In *Intl. Conference on Open Distributed Processing*, Australia, February 1995.

[15] S. Mullender. Kernel Support for Distributed Systems. In S. Mullender, editor, *Distributed Systems*, pages 385–409. Addison-Wesley, 1993.

[16] K. Nahrstedt and R. Steinmetz. "Resource Management in Networked Multimedia Systems". *IEEE Computer*, 28(5):52–63, May 1995.

[17] C. Nicolau. "An Architecture for Real-Time Multimedia Communication Systems". *IEEE Journal on Selected Areas in Communications*, 8(3):391–400, April 1990.

[18] OMG. *The Common Object Request Broker: Architecture and Specification.* Object Management Group, 1992. OMG Document Number 91.12.1, Revision 1.1.

[19] R. Orfali and D. Harkey. "Client/Server with Distributed Objects". *BYTE*, 20(4):151–162, 1995.

[20] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. "CHORUS Distributed Operating Systems". *Computing Systems Journal*, 1(4):305–370, 1988.

[21] D. Shepherd and M. Salmony. "Extending OSI to Support Synchronization required by Multimedia Applications". *Computer Communications*, 13(7):399–406, September 1990.

[22] A. Snyder. "The Essence of Objects: Concepts and Terms". *IEEE Software*, 10(1):31–42, 1993.

[23] R. Steinmetz. "Human Perception of Jitter and Media Synchronisation". To appear in IEEE Journal on Selected Areas in Communication, 14(2), February 1996.

[24] H. Stenzel, K. Kansy, I. Herman, and G. Carson. "PREMO: An Architecture for Presentation of Multimedia Objects in an Open Envirnoment". In W. Herzner and F. Kappe, editors, *Multimedia/Hypermedia in Open Distributed Environments*, pages 77–96. Springer-Verlag, 1994.

[25] A. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 1992.

[26] D. Terry and D. Swinehart. "Managing Stored Voice in the Etherphone System". *ACM Transactions on Computer Systems*, 6(1):3–27, 1988.