THE ANNOTATION OF CONTINUOUS MEDIA

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT AT CANTERBURY

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY.

By

Carlos André Guimarães Ferraz

September 1995

To Ana Maria, my wife,

and André, my son.

Abstract

In principle, the presentation of continuous media is time-dependent. Examples of continuous media are audio, video and graphics animation. This work is on the support for the annotation of continuous media, or the integration of voice comments with continuous-media documents like music and video clips. This application has strict synchronisation requirements, both with respect to the media involved and to user interaction. The application involves functions such as storage, management, control of GUIs, and of continuous-medium devices. These are realised by components which can be distributed across a network.

New models and architectures have been defined to enable open distributed processing of applications, that is, distributed processing independent of operating systems. Abstractions are provided, which facilitate the development of applications, and these execute supported by platforms that implement such open architectures. These architectures have been based on an object-based client/server model. Our work aims at exploring object-orientation, open distributed processing and some characteristics of continuous media, through the development and use of the proposed application.

The application is designed as a set of objects with well-defined functions and which interact between themselves. A distinguishing feature of the application is that it involves reusable components and mechanisms. For example, a mechanism, which enables components to control logical clocks and synchronise them, is incorporated in the application in response to its synchronisation requirements. The implementation is based on ANSAware, a platform that supports open distributed processing and allows distributed objects to bind to each other, to interact with one another, and to exhibit concurrent activities. The performance of the implementation is examined with respect to the application's response to user requests. Response times of operations such as play, pause, etc., are measured, and the final results are better than a defined maximum tolerance.

An analysis of the development approach is made with respect to support for real-time activities in the application, and to software reuse in the model proposed. This thesis concludes by reviewing the suitability of the object-oriented approach for the development of distributed continuous media applications.

Acknowledgements

I am very grateful to my supervisor, Professor Peter Linington, for his constructive criticism, guidance, understanding and attention during the whole period of this work. His experience was fundamental to improve style and correctness in the thesis.

Others helped to make my work easier. In particular, I would like to thank David Barnes and Li Ning, with respect to the audio sub-system, and Chris Scott, Mike Rizzo and Ian Buckner, in relation to ANSAware.

I specially thank my parents, José Martiniano and Maria do Céu. They gave me love, education, and moral support throughout my life. Their support has been fundamental to what I have achieved.

My family and I would like to thank my grandmother, and my parents-in-law, for their kindness and encouragement throughout this period. Here, our life was made easier thanks to many people, and I would like to mention George Justo, Eduardo Albuquerque, Judith Kelner and Sue Davies.

The unforgettable experience we had here was made possible by many friends and colleagues. Special thanks go to Paulo Cunha, Silvio Meira, Richard Shaw, Eduardo Vegas, Helena Rodrigues and Geraldina Fernandes.

I acknowledge the financial support of Fundação CAPES, Brazil.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter starts by showing the motivation to annotate continuous media. The annotation of continuous media is an application that involves media such as voice, audio or video, and presents strict synchronisation requirements. Multimedia applications generally require distribution support to run more efficiently. The main approach used for structuring distributed systems has been the client/server model, and object-oriented operating systems or intermediate platforms based on this model enable the development and use of distributed applications. In this sense, this introductory chapter discusses multimedia applications – in particular, those involving time-dependent media – distributed systems, and describes the objectives of this work.

1.1 Motivation

This work is about *continuous media*; Ferrari *et al* [Ferrari 92] 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)."

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

The power of audio and video in areas that provide information and entertainment has been demonstrated by, for example, radio and television. The existence of hardware and software components capable of handling digital audio and video makes computers, with their ability to provide interaction facilities, more and more useful in education, training, advertisement and many other areas. Further developments will allow a widespread use of continuous-media documents (i.e. documents in the form of audio and/or video) within and between organisations, improving and making more efficient methods of working based on, for example, teleconference and computer-supported cooperative work (CSCW).

The advantages of annotation of text documents, and more generally, static-media documents (e.g. images and graphics) are well established [Terry 88, Fish 88, Tilley 91, Trigg 88, Dewan 93]. It is also desirable to be able to annotate continuous media documents to help activities like analysis, revision, authoring and criticism – of music, natural sounds, speech, video, etc. Studies presented in [Chalfonte 91, Kraut 92] show that spoken annotations are more likely to be used to comment on higher-level issues in a document, adding features like explanation that make the comments more expressive. Van Ness [van Nes 92] describes human-factors research on annotation of (static) documents indicating a preference for voice annotations as being more efficient than text annotations. Similarly, annotation of continuous documents can be more efficient and more expressive using voice.

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 synchronise, as for example in the presentation of annotations and continuous documents. Synchronisation 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.

1.2 Multimedia Applications

Multimedia applications deal with various types of data such as text, video and audio, that are usually captured or presented simultaneously. In this sense, the annotation of continuous

media can be characterised as a multimedia application. The advent of multiple media has increased the expressiveness of applications, since information can be represented by appropriate types of media. Ghandeharizadeh and Ramos (in [Ghandeharizadeh 93]) say

> "Multimedia Information Systems have emerged as an essential component of many application domains... because these systems utilise a variety of human senses to provide an effective means of communicating information."

This shows that the meaning of things is captured rapidly, without much effort, because they are appropriately represented. For example, electronic messages can use voice, simulation of chemical experiments can use graphics, and so on. And these can be composed in multimedia applications enabling, for example, the simulations of chemical experiments to be accompanied by comments, either in the form of text or voice. These features and the possibility of providing user interaction make computer-supported multimedia systems widely applicable.

1.2.1 Types of Media

The term 'multimedia' concerns the integration of data representing distinct forms of information such as text, graphics, images, video, sound and voice. These are distinguished by their internal and external characteristics. One important aspect of multimedia systems is the representation of time dependencies of the data involved.

Time dependencies

In multimedia applications there can be composition of data presenting distinct time dependencies, as described in table 1 adapted from [Little 94]. An example is voice annotations

(i.e. *continuous* data) of *static* images. In addition to being continuous, the voice annotations can be *live* or *stored*, i.e. the same type of data may present more than one time dependency.

Type	Definition
Static	no time dependency
Discrete	single element
Transient	short-lived
Natural or implied	real-world time dependencies
Synthetic	artificially created time dependencies
Continuous	playout contiguous in time
Persistent	maintained in a database
Live	originated in real time
Stored	originated from prerecorded storage

Table 1: Time dependencies of data.

1.2.2 Constraints

Multimedia applications require handling of timing constraints [Stefani 92], which may be caused by

- the different types of representation media, i.e. types of data exchanged between application components at some presentation interfaces,
- synchronisation of media presentation, or
- interactivity.

The different media are presented or captured by devices which provide interfaces classified according to their behaviour as follows.

Discrete vs. continuous interfaces

Devices' discrete interfaces cause no timing constraints on the presentation of media data (presentation constraints are only defined by the application), whereas the behaviour of continuous interfaces imposes timing constraints (rate of presentation, in particular) on the capture or presentation of continuous media.

1.2.3 Distributed Requirements

According to Steinmetz in [Steinmetz 96],

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

Functional components, some in charge of specific media (e.g. an audio store), must interact to achieve the intentions of a multimedia application. The nature of this kind of application implies that several data streams have to be handled in parallel [Shepherd 90]. Appropriate distribution support allows application components to be designed for distribution across a network (see section 1.3.2).

The International Standards Organization (ISO) has been working on ways to allow multimedia presentations to be portable, distributed for example in CD-ROMs or over networks. The subcommittee ISO/IEC JTC1/SC29 (Coding of Audio, Picture, Multimedia, and Hypermedia Information) has three working groups:

- Joint Photographic Experts Group (JPEG), which defines the compression encoding of still images [ISO 92b],

- Moving Pictures Experts Group (MPEG), which defines a compression and interchange format for video, including audio [ISO 92a], and
- Multimedia and Hypermedia Information Coding Experts Group (MHEG) defines a model, which is expected to allow the description of the interrelationship between the different components of a multimedia presentation [Meyer-Boudnik 95].

Our work would benefit from the above for information distribution, but it is initially interested in the distributed *processing* of multimedia applications, specifically continuous-media ones.

1.2.4 Annotation and Related Work

An annotation is a note which is associated with a certain portion of a document for purposes like commenting, augmenting information or reminding the reader of something. Three basic elements are involved:

(a) the main document that is being followed by the reader;

(b) the annotation, itself a document that contains information referring to a certain portion of the main or underlying document; and

(c) the association between the two types of document.

These can be modelled by the *hypertext*[1] concepts of

- *node*: an information container (i.e. a document), and

[1] Hypertext is a model for establishing relationships between documents without a fixed sequential structure. See [Conklin 87] for a good introduction.

- *link*: a logical connection between nodes.

Another concept, *anchor*, an icon or other indicator that highlights the portion of a node that is linked to another node, can be very useful in applications involving annotations. A hypertext model for the annotation of documents is seen in figure 1. In fact, the suitability

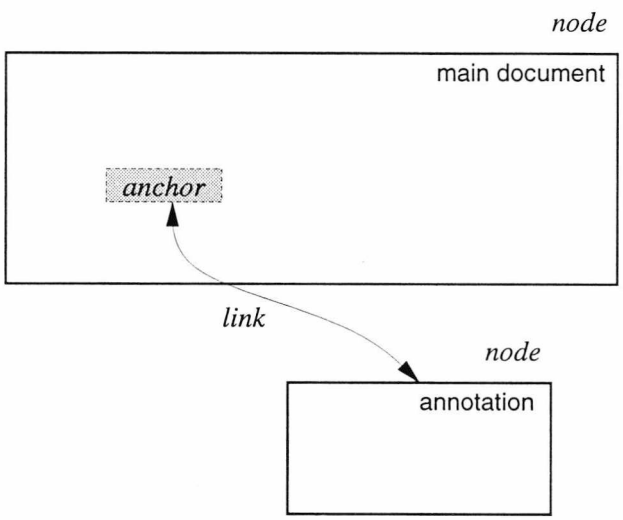


Figure 1: Hypertext model for the annotation of documents.

of this model for the annotation of documents is proposed, for example, in [Conklin 87]. The reason is that hypertext makes the creation of new references easy, which enables users to annotate documents (without changing them).

Hypertext can be extended to the more general concept of *hypermedia*, in which integrating elements can be text, graphics, sound, images, and video. The integration of information represented by multiple media basically follows the same model. This basic model can also be used when both information and management are distributed [Noll 91].

Audio, video and animation, that is continuous media, present new challenges for documents because they involve *time*. Given this, synchronisation becomes an important issue: multimedia events should happen in specific relationships with one another [Zellweger 92].

In the annotation of a video clip, for example, a comment about the jump of a horse that is jumping a fence should happen when the jump over the fence actually appears in scene.

Much work has been done on annotations [Thomas 85, Terry 88, Hsiao 89, Mackay 89, Dewan 93, Gintell 94]. These are divided into two classes:

(a) annotation of static media, and

(b) annotation of continuous media,

with annotations usually made of either text or voice.

The Etherphone system [Terry 88, Vin 91] allows embedding of speech in documents such as annotated manuscript or program documentation. It introduced voice annotations, which lead to a number of other pieces of work, like [Hsiao 89], that allowed voice annotation of still images. In an initial stage of our work, a voice annotator of clocked sequences of images was developed. Our work uses a distributed audio system [Li 94] that adopts the concept of *voice rope* introduced in [Terry 88].

Annotations are particularly useful in cooperative work. For example, Quilt [Fish 88] is a tool for collaborative document production using hypermedia links that allow people to attach text and voice annotations to the document. Also, Scrutiny[2] [Gintell 94] is a system for performing software inspection and review. In it, *inspectors* examine items to find defects and are allowed to enter textual comments and questions (*annotations*). Each annotation refers to a specific portion of an item and may be referenced via a hyperlink. They may be private or shared, and annotations may reference existing ones.

EVA, an experimental video annotator [Mackay 89], is a study on the annotation of video using symbols (or single-word annotations) to help video analysis. The annotations

[2]Scrutiny is a trademark of Bull HN Information Systems Inc.

are tags that help users search for segments (in a video) which are associated to specific tags representing their interests.

If both the base document and the annotation are continuous, there is a much more strict demand for synchronisation. Voice and the annotated media (audio or video) can occur simultaneously, and tolerance for time discrepancies between an annotation and the document portion to which it refers is a *quality of service* parameter – *maximum jitter*, i.e. the maximum acceptable time-difference between corresponding segments of synchronous streams. Stream synchronisation has been studied in detail, as in [Little 90], which considers intermedia timing in multimedia composition, and [Steinmetz 90], which addresses synchronisation mechanisms' characteristics. Careful management and control of time are necessary, as in the case of separate streams that require a buffering scheme to maintain the synchronisation of each individual stream over short periods of time and that the application ensures that they remain synchronised with respect to each other over longer periods of time [Nicolau 90]. In the Amsterdam Hypermedia Model [Hardman 94], time and the concept of *context* are added to the Dexter hypertext model [Halasz 94], allowing concurrent presentation of linked nodes.

1.3 Distributed Systems

People work exchanging information between each other, so that it is natural to think about the need to make the computers they use exchange information too [Khoshafian 92]. In [Schroeder 93] the author defines the ideal distributed system as a combination of the advantages of centralised systems (stand-alone personal computers or mainframes) with the advantages of networked systems (a collection of workstations/personal computers and

servers connected by a communication network), plus real security and high availability. Real security is not offered by centralised systems, which have a single security domain, because a fault can be exploited to break the security of the entire system. On the other hand, networked systems do not offer real security for having multiple security domains, which makes security control extremely difficult. Both types of systems have their advantages. Equally in relation to high availability, there are pros and cons from both sides (see details in [Schroeder 93, *pp. 3-4*]).

The advantages of centralised systems are given as

- accessibility: all information and resources are equally accessible;
- coherence: functions work the same way and objects have the same name everywhere in a centralised system; and
- manageability: a centralised system is easier to manage.

The advantages of networked systems are

- sharing of information and resources spread geographically,
- use of small, cost-effective computers,
- incremental growth, i.e. computing power can be added in small increments, and
- autonomy.

Nevertheless, in order to construct a distributed system, it is necessary to define the interconnection of the system's components and how it should be structured. It is also desirable to have means for the development of distributed applications.

Recognising the need for communication between different computers, ISO has proposed a collection of protocols in a reference model called Open Systems Interconnection, or OSI [ISO 81, ISO 84]. This reference model, where lower layers provide service to higher ones, has been used as an architectural model for many purposes (e.g. applications design). Not all the layers have to be used in the interconnection of systems. Most LAN[3]-based distributed systems use only a subset of the protocol stack for efficiency-related reasons, since a significant overhead is imposed by the addition of headers in all layers when a message is sent and removal of the headers when it is received.

The OSI Reference Model is limited to peer-to-peer communication. It is defined relative to the physical points of interconnection, abstracting communication from the internal structure and behaviour of the systems involved [Herbert 89]. The model was not intended to provide standards for distributed computing [Mullender 89]. Additionally, the OSI standards do not define facilities, such as synchronisation, required by distributed multimedia applications [Shepherd 90].

1.3.1 The Client/Server Model

The client/server model is used as an approach for the structuring of distributed systems. It says a distributed system should be organised as a group of cooperating processes, called *servers*, that offer services to other processes, called *clients* [Tanenbaum 92]. Clients and servers usually share responsibilities – for example, a client focuses on the presentation of data to the user, while a server concentrates on the storage and retrieval of the data. In the example of figure 2, a client and a server communicate through the use of *request* and *reply*

[3]LAN = Local Area Network.

messages. Basically, five steps occur:

1. the client makes a service request,
2. the server receives the request,
3. it is processed,
4. the processed results are returned, and
5. the client receives the reply.

The client/server approach is used both as a model for structuring distributed computer systems and as a model for distributed computing.

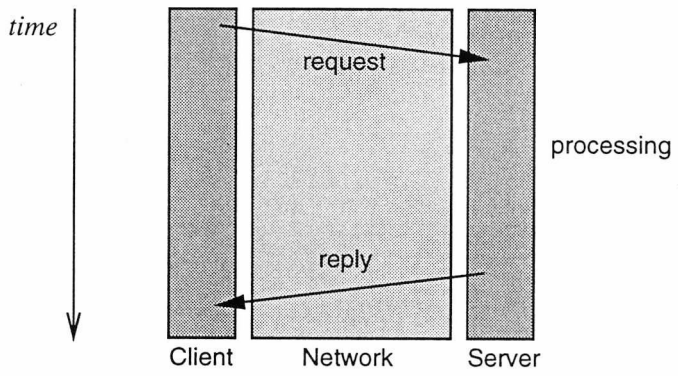


Figure 2: Client-server communication.

The client/server model is the most used model for distributed computing. The major technologies capable of creating client/server applications, and which provide tools and middleware for so doing, are SQL[4] databases, TP[5] monitors, groupware and distributed objects [Orfali 95b].

[4]SQL = Structured Query Language.
[5]TP = Transaction Processing

SQL manages data and the functions that manipulate them, via *stored procedures* which are composed of SQL statements and procedural logic, and are stored in a server database. It is said to be easy to create client/server applications in single-vendor/single-server environments, given the availability of tools.

TP monitors manage processes and orchestrate programs by breaking complex applications into pieces of code called *transactions*, which bind clients and servers. By putting themselves between clients and servers, TP monitors can manage transactions, route them across systems, load-balance their execution, and restart them after failures.

Groupware is a model of computing which helps users collect unstructured data and organise them as a collection of documents, via technologies such as multimedia document management, work flow, eletronic mail, conferencing, and work scheduling. It enables users to view documents, and to store, replicate and route them over a network. Groupware manages document databases in a client/server fashion.

Distributed objects encapsulate data and logic, and can be placed anywhere on networks, run on different platforms, and manage themselves and the resources they control. This technology is discussed further in Chapter 2.

Examples of network servers

The approach used in many network systems is based on the client/server model. Some examples are print services, database servers and file systems. They all have the objective of providing services to multiple users. Print services give users access to printers, allowing those to send files to be printed, to cancel printing, etc. Database servers are responsible for maintaining information bases. They provide concurrent access to the bases, and maintain the consistency and validity of the data.

File systems are particularly important applications of the client/server model and are discussed in more detail. The most popular distributed file system is the Network File System (NFS) of Sun Microsystems, which allows a client workstation to perform transparent file access over the network. It was designed to be machine-, operating system-, network-, and transport protocol-independent. Client file access calls are converted to NFS protocol requests. The server receives the request, performs the actual file system operation, and sends a response back to the client, as in figure 2. NFS uses *remote procedure calls* (RPC) for the exchange of commands and data with a remote server. RPC is an abstraction that makes a server appear to be one function call away; its primitives are built on top of the external data representation (XDR), which provides a machine-independent method of representing data. Tanenbaum [Tanenbaum 92, *Chapter 13*] distinguishes between *file service* and *file server* saying that in effect, the file service specifies the file system's interface to the clients, whereas a file server is a process that runs on some machine and helps implement the file service. A system may have one file server or several, and in a distributed system, the clients should not need to know the number of file servers and the location or function of each one of them. He gives as an example of a well-structured distributed file system, the Andrew File System (AFS) of the Carnegie-Mellon University, which works by caching whole files on client disks, uploading them when they are closed.

1.3.2 Development of Distributed Applications

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 (middleware) 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.

The basic needs for distributed applications are support for communication, concurrency and synchronisation. 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 synchronisation (see section 2.4) – 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 [Mullender 93b].

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/ORB (Object Request Broker) insulate clients from the mechanisms used to communicate with, activate, or store server objects [Orfali 95a]. 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 the following chapter.

1.4 Objectives

Ordinary users do not run operating systems, but applications. An operating system is an intermediate software layer that hides hardware details and manages system resources [Johansen 94]. In networked systems, there is distribution of both hardware and software components, and distributed systems can differ from networked systems through the property of transparency, which makes them look, to users, as though they were centralised. With the advent of distributed operating systems, such as Amoeba [Tanenbaum 90], Chorus [Rozier 88] and Mach [Accetta 86], and open distribution platforms like ANSA [APM 93a], CORBA [Orfali 95a], and DCE [Johnson 94], which are capable of providing independence from underlying systems, distributed applications can become reality. Applications already run on centralised systems and on networked ones, but their migration to the kind of platform provided by distributed systems is a matter for further experiment [Johansen 94].

The primary objective of this work is the development of a distributed continuous-media application, considering the use of digital continuous-media documents in networked environments and the challenges imposed by continuous media, in particular, requirements for data transmission and synchronisation. For the development of the application, an object-oriented method, based on the client/server model, will be used, and the application is built on top of a distribution platform. Support is needed for aspects such as

- construction of the application, concentrating on programming and integration support, including the use of existing services and creation of new ones;
- continuous media, that is, types support and data transfer;

- concurrency, i.e. processing support for execution threads within processes – scheduling;
- synchronisation, in terms of supporting mechanisms;
- reusability of code, components and mechanisms.

Where direct support is not provided by the distribution platform, we will use the support available to develop the required mechanisms.

The resulting Annotator should be a component that can be integrated in different applications to provide the facility of document annotation. However, the principal contribution of this work is concerned with the development of distributed applications in general. Focusing on the most strict requirements put on support platforms, such as those imposed by multimedia applications involving continuous media, this work is to explore as many features of the platforms as possible. The results include the description of the development experience and the indication of points that may need some improvement.

1.5 Thesis Outline

The thesis describes the approach used to design, implement and measure the performance of the distributed application involving continuous media. This chapter was concerned with providing the motivation to annotate continuous media, related work and general discussions about multimedia applications and distributed systems. The following chapters build on this introduction.

Chapter 2 is concerned with support for distributed applications. Development and use of distributed applications have been based on the object-oriented approach. Object

orientation is introduced, with some concepts and benefits being highlighted. The approach has been particularly used in the distributed systems domain, involving operating systems, intermediate platforms, applications and environments. A discussion on the emerging standards for open distributed processing is included, and the ANSAware platform is introduced. ANSAware's support to distributed applications is presented and discussed.

Chapter 3 presents the design of the application regarding the annotation of continuous media. The application is described as consisting of an annotator and a continuous-document presenter, such as a music player, a video player, etc. The required support, in terms of storage, database management and continuous-media devices control, is discussed and the modules of the application are presented. The application has to perform according to some requirements such as bandwidth, synchronisation and response time. These are discussed considering that the application involves continuous media and user interaction. And finally, the environment in which the application is developed and used is discussed together with its limitations.

Chapter 4 discusses the implementation of the application. A music player is considered as the annotator's partner, and so the implementation of the annotation of music is presented. Initially, the possible types of annotation and the relationships between annotations and the presentation of the base document are defined. The discussion involves user interaction, including manipulation of presentation rates and its synchronisation effects, considering the GUIs for the annotator and the music player. Then, the integration of the application components, including the definition of their interfaces, is presented.

Chapter 5 is concerned with the performance of the implementation. Response times for user requests made via the GUIs are measured, allowing the synchronisation between the application components involved in the actions to be examined. Satisfactory results are achieved after three stages of modifications. The method of making the measurements, the factors that affect the application's performance and the results of the modifications made are discussed in this chapter.

Chapter 6 analyses the approach used to model distributed continuous media applications in terms of an object-based description of support for continuous media and of software reuse. Support for continuous media is discussed in terms of rate control and stream handling, and it is compared with a different approach. The discussion about reuse involves the reuse of interface specifications, code reuse, and the reuse of the component objects presented in the model.

Chapter 7 concludes this thesis by presenting a summary of it, considering plans for future work, which include enhancements in the application and porting it to another platform with a similar object model, and drawing final remarks.

Chapter 2

Background

Here the background for the definition of distributed applications is given. Object-orientation is discussed and some of its concepts are introduced. Then, it is related to distributed systems, and work on the definition of architectures that aim at supporting the development and use of object-based distributed applications in open environments is highlighted. Finally, a distribution platform, called ANSAware, is presented.

2.1 Introduction

The purpose of this chapter is to present the basic technologies that support our work. The development of distributed applications needs support in terms of modelling techniques and architectures that describe means for distribution and interaction of application components. One of the most powerful models for systems development is based on entities called *objects*, which resemble real-world objects, as they can be *active* or *passive*, and have *interfaces* to interact with the environment.

The object-oriented approach, or object-orientation, to be more generic, has been used in a large number of areas of computing. It has been used to develop information systems, control systems, etc. In particular, object-orientation has been involved in aspects of distributed systems, from concepts to design, programming, and support. There are object-oriented distributed operating systems, intermediate platforms to support distribution based on objects, and object-oriented distributed applications.

The following sections of this chapter highlight the aspects of object-orientation related to distributed systems, initially presenting a general overview of it, and work on distribution architectures, designed to support development and use of distributed applications. There is no intention to cover all the aspects of any of the topics, mainly because of the wide spectrum of them. Some more detail is given, however, with respect to the particular distribution platform used for the application described in this thesis (**the annotation of continuous media**).

2.2 Object-Orientation

Object-orientation is not restricted to programming languages, databases or software engineering; thus, it may be interpreted in many different ways. *Objects* have different roles in the variety of different areas where they are applied. For example, in a concurrent programming environment, objects need to be *active objects* with a queueing facility for incoming messages, whereas database systems interpret objects as *imperative objects* with a unique identity determined by their attributes and operations.

One observes that objects may be of different kinds for different purposes. Wegner [Wegner 90] classifies them as

- functional objects: a functional object has no state and no separate identity, and operations are invoked by function calls. Examples of functional objects are the objects TRUE and FALSE of the *class* BOOL with operations such as `and`, `or` and `not` in a functional programming language;

- imperative objects: imperative objects have the property of being passive until they are activated by a received message; and

- active objects: these may be executing when a message arrives. Incoming messages may have to be queued and to wait for execution. The possible states of an active object are idle (sleeping), busy (executing) and blocked (waiting for resources or completion of subtasks). Active objects are applied, for example, in concurrent object-oriented programming [Lea 93, Meyer 93, Karaorman 93].

Objects interact by sending messages to communicate with each other. The following three components of a message are identified:

1. Receptor, which specifies the object to which the message is sent.

2. Selector, which specifies the appropriate *method* which should be invoked as a result of the message.

3. Parameters, which are function arguments or operands needed by the method associated with a specific message.

If an object receives a message, it usually reacts by executing a method (or *operation*), but another possible reaction is an *exception*, i.e. an error routine.

In order to enable object interaction, each object provides an *interface* (or protocol). The interface determines which parts (e.g. methods) of an object are accessible from other objects.

2.2.1 Concepts

The following paragraphs describe some of the concepts of terms particularly associated with object-orientation. Some terms have already been introduced, and only the ones that are in the scope of this work are mentioned. These are:

- objects and interfaces,
- encapsulation and abstraction,
- method (operation),
- class,
- inheritance, and
- binding (static and dynamic).

Object. An object is usually seen as a representation of an entity. Each object in a certain environment has a unique *identifier*, an updatable *state*, and *behaviour*. Unique object-identifiers distinguish objects from each other without the need to compare their values or their behaviour.

Another point related to object identity is concerned with sharing of an object by multiple clients. Sharing of object means that two or more references can point to the same

object. Updates on the object pointed to can be seen from the clients, so only one update is necessary.

Encapsulation. Encapsulation is defined by Booch [Booch 91, *p. 46*] as

> "... the process of hiding all of the details of an object that do not contribute to its essential characteristics."

He also says that encapsulation and *abstraction* are complementary concepts, with abstraction focusing on the outside view of an object and encapsulation (or *information hiding*) preventing clients from seeing its inside view, where the behaviour of the abstraction is implemented. In object-oriented systems, objects are the units of encapsulation (modules). An object encapsulates state (data) and behaviour (operations). The operations are the means to manipulate the state.

Method. A method provides the implementation of an operation on an object. Different objects may have different methods for the same operation. Bertino [Bertino 91] identifies the two constituent parts of a method as follows:

(a) Method signature. The signature consists of the method's name, the names and types of the parameters, and the types of the results.

(b) Method implementation. The implementation of the method is written in a programming language and is executed when an appropriate message is received.

Methods can have

- a return value. Those which do not have a return value are only used to manipulate the state of objects. Some methods manipulate the state of objects and have a return

value, whereas others have a return value and do not manipulate state, being only used to retrieve information about objects – these are called *retrieval methods*;

- side effects. Manipulation of the state of objects generates side effects. Retrieval methods, therefore, have no side effects; and
- parameters. If a method has parameters, there can be different results for each possible combination of parameters.

Class. Objects that have the same structural and behavioural characteristics – they can be distinguished by object identity only – can be grouped into a class. Classes are similar to types; thus, relationships can be established between classes, as well as between types.

Inheritance. The description of inheritance is valid both for classes and for types. Inheritance means that classes (types) can be derived from other classes (types). A superclass-subclass (supertype-subtype) relationship is established. The subclass (subtype) inherits the attributes and the operations of the superclass (supertype). The mechanism that allows a subclass (subtype) to define additional operations and attributes is called *specialisation*. On the other hand, instances of the superclass (supertype) generalize (restrict) the instances of the subclasses (subtypes) – this mechanism is called *generalisation* [Banerjee 87].

There can be

- inheritance of implementation, which is used to facilitate the implementation of classes by using code of existing classes – a subclass relation is established; and
- inheritance of behaviour, used to establish an "is-a" relationship between objects. This is called subtyping.

Subtyping occurs if the properties of a type are inherited by a subtype. *Subclassing* means that not only inheritance of specification but also inheritance of implementation occur. In class-based object-oriented systems, a subclass inherits the operations implemented for the superclass. Since the subclass is a specialisation of the superclass, it can also have a specialised version of the existing operations of the superclass. Thus, it is possible for a subclass to *override* the operations of the superclass.

Binding. The time when the code is bound to a method signature determines two kinds of binding:

- *Static binding* (or early binding). Early binding means that binding occurs at compile time.
- *Dynamic binding* (or late binding). Late binding means that binding occurs at run time.

Dynamic binding is needed, for example, in the situation in which the appropriate code for a method is not known at compile time. The compilation can proceed given the possibility of late binding.

2.2.2 Object-based vs. Object-oriented

The terms 'object-based' and 'object-oriented' are often used interchangeably. When there is concern with the use of the appropriate term, 'object-based' and 'object-oriented' are normally distinguished considering the presence or absence of inheritance. Booch [Booch 91], Cardelli and Wegner [Cardelli 85, Wegner 87] define 'object-based' as 'object-oriented' without inheritance, which means that object-based systems have greater freedom

to select an appropriate kind of binding to support distribution, for example.

2.2.3 Benefits

Among the benefits or advantages of object orientation, perhaps the most important ones are

- naturalness: the concept of object appeals well to human cognition;
- a strong relationship with the real word in terms of modelling;
- reusability of software and design;
- encapsulation and abstraction.

Other additional advantages [Booch 94] include

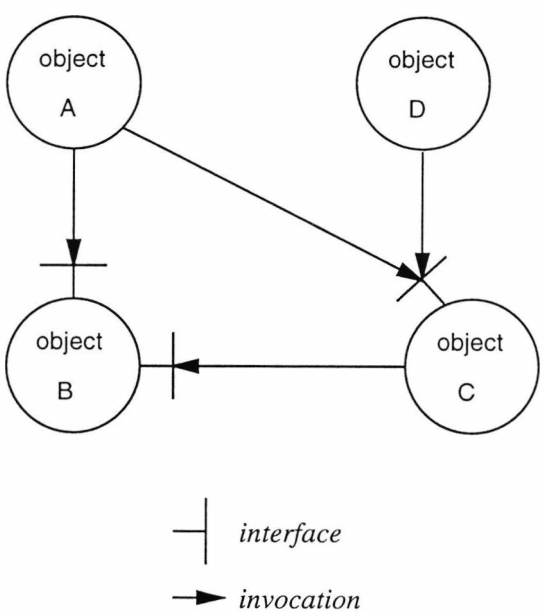
- faster development of systems,
- easier maintenance,
- evolution, i.e. systems may have more ability to evolve, and
- better quality.

2.3 Developing and Processing Object-Based Distributed Applications

Object orientation has been used in many domains, notably in programming languages and databases. Also a number of techniques have been based on it to help analysis,

called object-oriented analysis (OOA), design, called object-oriented design (OOD), and programming, known as object-oriented programming (OOP). 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 behaviour and offer interfaces for interaction, appears both natural and appropriate. In figure 3, 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.



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

Figure 3: Object model.

The following concepts, identified and discussed in [Snyder 93], confirm their appropriateness for use in distributed systems:

- All objects embody an abstraction.
- Objects provide services.
- Clients issue requests.
- Objects are encapsulated.
- Requests identify operations.
- 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 [Rozier 88]), intermediate architectures (e.g. CORBA [OMG 92, Orfali 95a] and ANSA [APM 93a]) and applications and environments, such as PREMO [Herman 94, Stenzel 94], a presentation environment for multimedia objects under development by ISO/IEC JTC1/SC24/WG6 [ISO 95a]. In connection with their use in distributed systems, object-oriented techniques have been applied in the construction of

telecommunication software, given the demand for the quick development of new services [Yamazaki 93] – here the issues include maintainability, reliability and performance – and parallel systems [Grimshaw 93], in which performance is the major issue and a hospitable programming environment is desirable.

Operating systems. Object-oriented operating systems provide resources through objects that can be distributed. Additionally, objects can be used to model both the hardware interface, the application interface, and all operating system concepts including mechanisms, policies and, as mentioned, resources [Campbell 93].

Distribution architectures. Distribution architectures describe the mechanisms by which objects make requests and receive responses in a transparent fashion. These intermediate platforms are built to support distributed applications, handling distribution of messages between application objects, and aim at providing abstraction. The benefits include enhanced application portability, ease of application distribution and reconfiguration, ease of application integration, and ease of software reuse [Burleigh 93].

Applications and environments. These are usually built on top of distribution platforms, particularly in an intermediate layer to allow for, e.g. portability and interoperability – *open systems*. The application on **the annotation of continuous media** is built on top of an object-based distribution platform and its development and performance are discussed in the following chapters (3, 4 and 5).

Applications are getting larger and more performance-critical, and some of their components may be distributed across heterogeneous networks. The trend for open, distributed, object-oriented computing is represented by efforts being made by the Object Management

Group (OMG), an industry consortium, and by ISO in order to develop standards to enable the interconnection of heterogeneous distributed applications – object-oriented techniques were chosen to describe the models.

2.3.1 Emerging Standards for Open Distributed Processing

The two main groups, the ISO/IEC JTC1/SC21 WG7 and the OMG, searching for an architecture that allows the construction of system-independent distributed applications, have contacts with each other for the establishment of the necessary standards. Distribution platforms try to conform with the emerging standards discussed here.

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 [OMG 92] 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 [Orfali 95a].

ODP

The ISO's Open Distributed Processing Reference Model (ODP-RM) [ISO 95b] 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. According to the ODP model, a system is considered from different *viewpoints*, which reflect specific design concerns [Linington 92, Linington 95]. 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.

2.4 ANSAware

ANSAware is a platform that follows the ODP reference model. It is an implementation of ANSA (Advanced Networked Systems Architecture), an architecture for open distributed processing which supports the design and construction of distributed applications, and is not constrained by network structure, or heterogeneous hardware and operating systems [APM 93a]. There are ANSAware ports to systems like UNIX[1], VMS, MS-DOS and Chorus, for example. 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

[1]UNIX is a registered trademark of AT&T Bell Laboratories.

several interfaces, each offering the same or different services – 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 [APM 93b]. (There is also dynamic instantiation of interfaces, discussed on page 38.)

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 synchronisation 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 [APM 93c]. Components have multiple threads and may optionally support multi-tasking.

ANSAware allows concurrent activities within capsules, and thus, provides an inter-task synchronisation 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.

The following algorithm makes sure that the lines and columns of a matrix are drawn at the same colour:

```
            /* thread A */

for (colour=0; colour<256; colour++) {
    for (i = 0; i < 5; i++) {
        draw_line(colour,i);
    }
    ecs_advance(five_line_count);
    ecs_await(five_col_count,
            ecs_ticket(five_col_seq));
}
```

```
            /* thread B */

for (colour=0; colour<256; colour++) {
    for (i = 0; i < 5; i++) {
        draw_column(colour,i);
    }
    ecs_advance(five_col_count);
    ecs_await(five_line_count,
            ecs_ticket(five_line_seq));
}
```

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 4 shows the use of trader's service and non-traded interfaces (the example interface (`ExampleIntf`) given before is considered as s1).

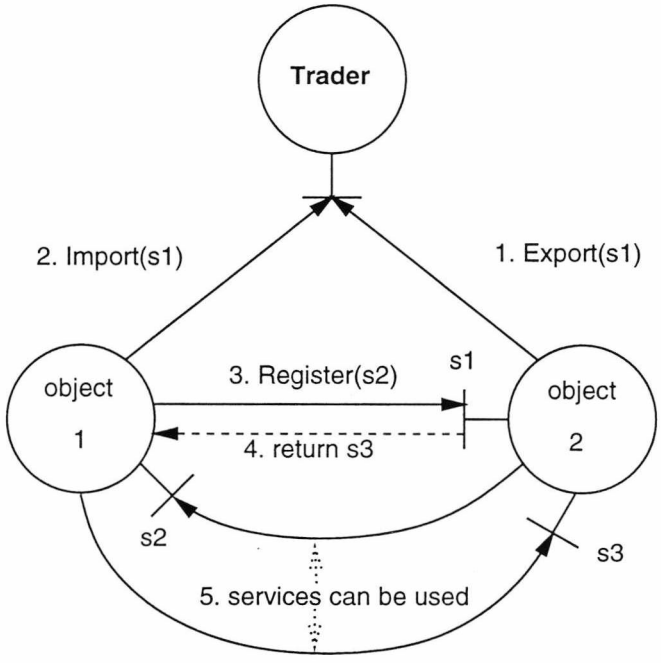


Figure 4: Traded and non-traded services.

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 [Tanenbaum 92, *pp. 420–427*]. See figure 5 for an example of the use of stubs.

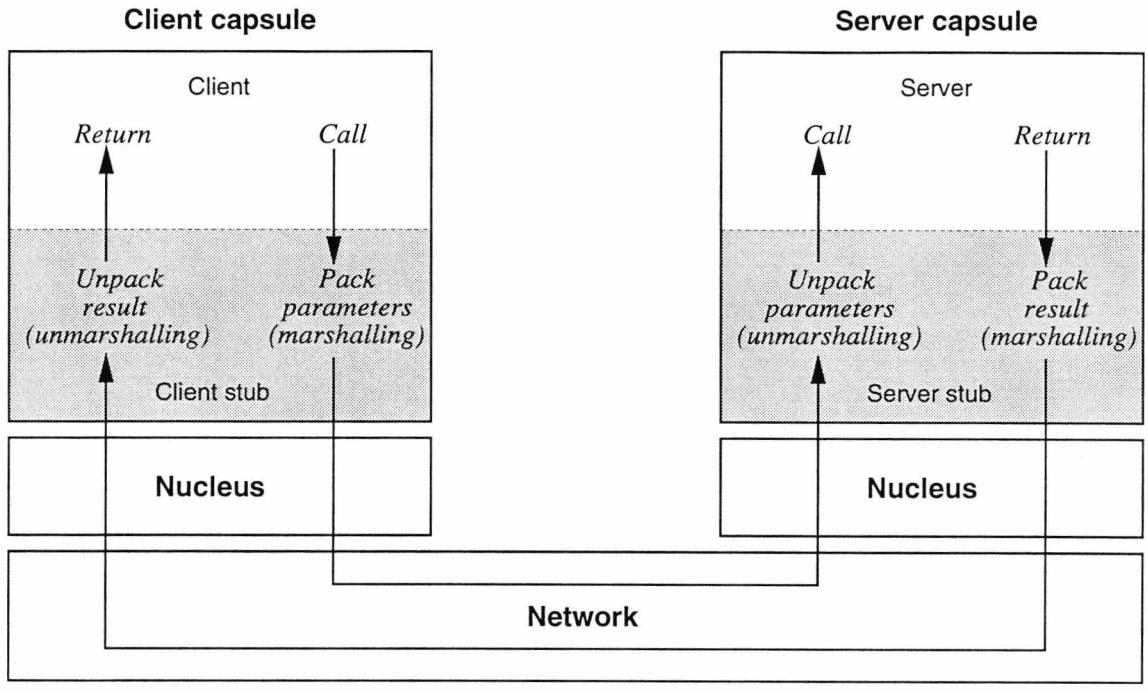


Figure 5: RPC transparency.

2.5 Conclusion

This chapter has identified the suitability of object-oriented techniques for the design and development of distributed systems. These are modelled as a collection of objects that are able to send and receive messages to each other. Their external behaviour is specified at interfaces, which they make available to represent the services they can provide. Services can be shared by various objects, so that updates on the data managed by the object that provides a certain service need only to be made once, as they can then be seen by all the objects that share the service. Interfaces provide an abstract view of the operations that can be used to perform services. The operations' actual implementations, called methods, are hidden, or encapsulated, by the objects that provide the services. Objects can be classified

according to their structural and behavioural characteristics. One of the benefits object-orientation can provide is the reusability of objects in systems and applications, added to other benefits that allow faster development, easier maintenance and extension of systems.

Because of their ability to provide services, abstractions, and interaction, amongst other features, objects have been applied in the development and use of distributed operating systems, intermediate architectures, applications and environments. Major work has been carried out on the definition of architectures to support the development of distributed applications in open environments. The efforts include the provision of transparencies, so that development and use can be made without concerns about, for example, access methods for and location of objects, and provision of consistency in the relationships between the human interface, the programming interface and communication protocols.

If architectures are implemented as intermediate platforms, they enable applications to be independent of the operating systems on which these distribution platforms are based. These platforms use the facilities given by the lower-layer systems to provide services to support the distribution of applications. If, for example, the operating system provides services such as concurrency support, etc., the intermediate layer needs only to provide the necessary system-independence without imposing much additional overhead.

ANSAware, the distribution platform used for our application, has been introduced. It provides services, such as trading, to allow application objects to export and import services in the form of interface references. Interface instances can be dynamically created and destroyed, and objects that possess references to them may request the services they provide. It is also possible to instantiate objects dynamically through the combined services provided by the trader and other architectural servers called node managers and factories. ANSAware supports transparencies such as location transparency and access transparency.

In ANSAware, the so-called computational objects are described using an interface definition language (IDL), for the interfaces, and a language that allows the embedding of invocations of interface operations in source code described in a standard programming language like C. Computational objects are compiled into engineering objects called capsules. A capsule is a protection domain and an atomic unit of failure; it can communicate with other capsules and can be multi-threaded, i.e. there can be concurrent activities within it. Thus, ANSAware supports concurrency and provides a mechanism for inter-task synchronisation. Communication between capsules is done via either asynchronous or synchronous RPC.

In the following, the object-oriented paradigm and ANSAware, as an object-based open distribution platform, are used for the development of an application that is interactive, needs synchronisation, and is composed of a number of communicating objects, some of them reusable.

Chapter 3

The Design of a Distributed Application

This chapter presents the design of a distributed application which makes possible the annotation of continuous media documents (eg. music and video clips) using voice. The composition of the continuous media involved in the application is discussed, with emphasis on the aspects of synchronisation, including real-time requirements. The modules of the application are presented, considering the needs for data storage, database management and continuous media devices control. The environment for the development of the application is presented and its limitations are discussed.

3.1 Introduction

With the advent of languages (e.g. LOTOS [Bolognesi 87]), environments (e.g. CONIC [Magee 89]) and, particularly, platforms (e.g. ANSAware [APM 93a] and Orbix [IONA 95]) to support distribution transparently, it has become easier and simpler to develop distributed systems. In multimedia applications, the media involved require different treatment in terms

of storage, interaction, transmission and representation. Modularity, concurrency and synchronisation are some of the key requirements of these applications. Implementation of the modules as independent processing elements that cooperate and communicate to achieve common goals can provide efficiency, especially when they run on different processors.

Advances described in [Mullender 93a, Tanenbaum 92, Fox 91, Liebhold 91], referring to technologies involving communication, operating systems and data compression (including the integration of de/compression facilities with presentation devices), allow the construction of multimedia applications with real-time requirements, or at least with fast response times. Real-time systems can be classified as *hard* and *soft* [Panzieri 93]. The latter are defined as real-time systems in which the ability to meet deadlines is required, but failure to do so does not indicate a system error.

Application design is the act of discovering the structure of an application and defining the modules of which it will consist [Lorin 90]. Functions are identified and grouped into modules according to the primary interests of the application, i.e. performance, extensibility, recovery, etc. Design has to do with the specification of computations within the modules and communications between them [Singhal 91].

The advent of multimedia has increased the possibilities of user interaction. Not only in terms of devices (screen, headphone, etc.), but also with respect to access forms (random access, browsing, etc.) in composite documents. Attention has to be given to observe constraints imposed by the components of the documents [Stefani 92].

In this chapter we present an application that is modular, interactive and is designed to meet real-time requirements. The design will also address, to a certain extent, issues such as generality, i.e. applicability to different media types, and extension of the functionality to accommodate group cooperation, following the fundamental principles for the design of

open distributed systems of orthogonality (independent definitions specifying independent architectural requirements), generality (preference to generic rather than special-purpose definitions) and open-endedness (maintenance of designs, i.e. extension and modification) [Vissers 91].

3.2 Description of the Application

The annotation of continuous media is an application that 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 and others), the reference chosen to link annotations and the respective document-segments is *time*, enabling synchronisation between them. Figure 6 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 synchronised, giving significance to the contents of the annotation.

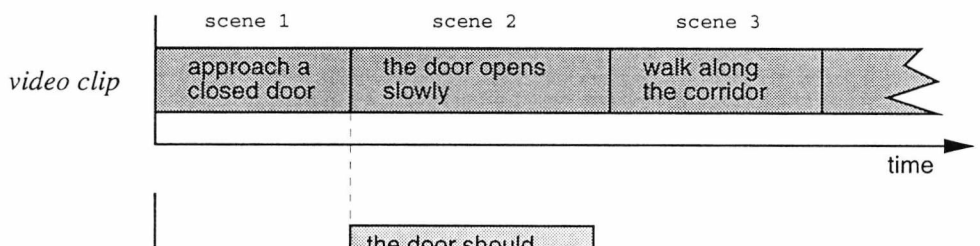


Figure 6: Example of an annotation.

The application could be used in areas involving film production (advertisement, cinema, etc.), arts criticism (music, cinema, etc.), among others, which require a close relationship

between remarks and the parts of the material under observation to which they refer. 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 an appropriate graphical interface including a timeline, giving users access by logical time so that it can be manipulated, avoiding the use of more complex structures. Furthermore, the composition of annotations and documents is transparent to the users; that is, they do not need to know that a voice-annotation and a video-document are stored separately – they are automatically associated by the application.

The design of the application draws on the *constructive approach*[Kramer 90, Kramer 93], which proposes steps that can be used recursively to build distributed systems. Notably, the first step, *structure and component identification*, which takes into account functional decomposition and entity modelling/identification of component types, is used here. Other steps, such as *interface specification* and *component elaboration*, i.e. functional description of behaviour, are used in Chapter 4, which discusses interaction in more detail.

3.2.1 Two Sub-systems

The application consists of two concurrent modules: one controls the presentation of continuous medium documents and the other enables the annotation of such *base documents*. The separation allows the annotator module to connect, in different instances, with presentation modules that support different types of documents, like music, video, animation, etc.

Control buttons allow the user to view presentations interactively rather than linearly. Through the interface of the annotator the user can request recording and playback of annotations. The two modules have to provide logical clocks and exchange information in order to keep them synchronised. The initial design structure of the **annotation of**

continuous media application can be seen in figure 7.

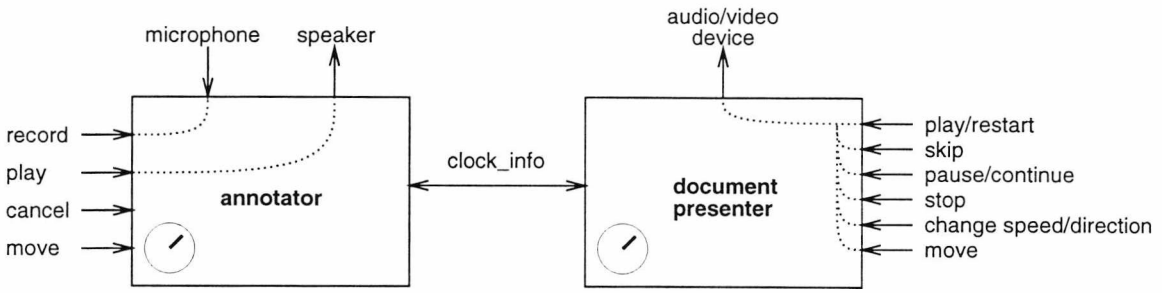


Figure 7: The two sub-systems of the application.

For the presentation of both base document and annotations, and for the recording of annotations, the application needs support for data storage, database management and device control for the media involved.

3.2.2 Support

Storage

Continuous media are stored in various standard-formats (.au, .avi, etc.), which cannot be mixed with other types of information. The storage requirements of the application are classified as

- **annotation-information storage**: the application is supposed to provide storage for structured information about annotations, e.g. records containing *time-references*, etc., and

- **continuous-media storage**: specialised by media type – video store, audio store, etc. Note that storage and retrieval of the voice part of an annotation should be handled by an audio store.

The identification of the general requirement for data handling leads to the first step in the decomposition of the annotation and presentation modules, as shown in figure 8 and figure 9, respectively.

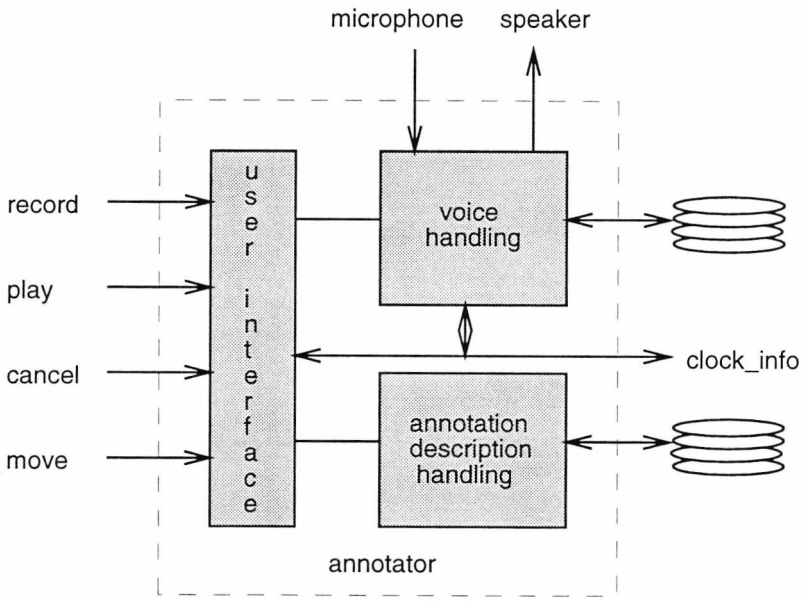


Figure 8: Annotator decomposition.

Management

The stored data must be organised so that references can be maintained and the client modules can refer to the data abstractly. Database facilities, supporting search, add and delete, are provided for this purpose.

The application has to be capable of incorporating modules that support structures which represent annotations, audio documents and video documents, depending on which types of media are involved in the *base documents*. Annotations are compound structures consisting of information, which describes the relationships between them and base documents, and voice, an audio document, which contains user-remarks on segments in the base

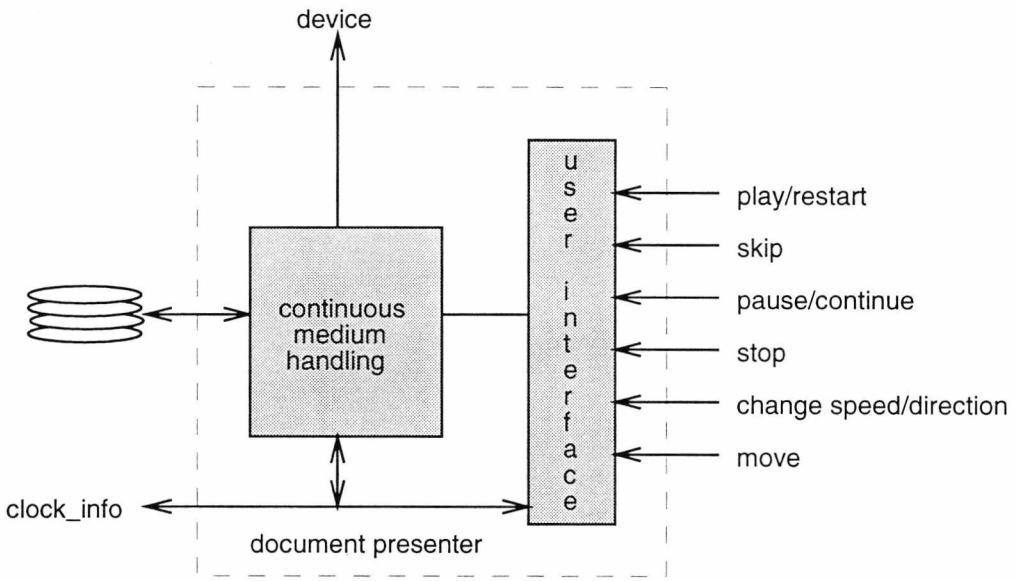


Figure 9: Presenter decomposition.

documents. The voice part of annotations is managed by an audio database module, and the reference-information, which includes the reference to the voice document, is managed by the **annotations database manager** module.

Besides the usual database operations, the management of continuous media requires operations to record, play, stop and build sequences of stored audio and video. In fact, abstractions for continuous media could be treated uniformly by a database manager without compromising the different requirements that the actual audio and video data may have (the data are manipulated by the respective storage modules – audio or video). Therefore, the general term *continuous-medium ropes*, from the *voice ropes* defined in [Terry 88], is appropriate to represent both audio and video sequences. Data referenced by continuous-medium ropes can be stored in distinct files, providing more flexibility in terms of data composition to application users. A clear distinction between the management/manipulation and storage functions is observed.

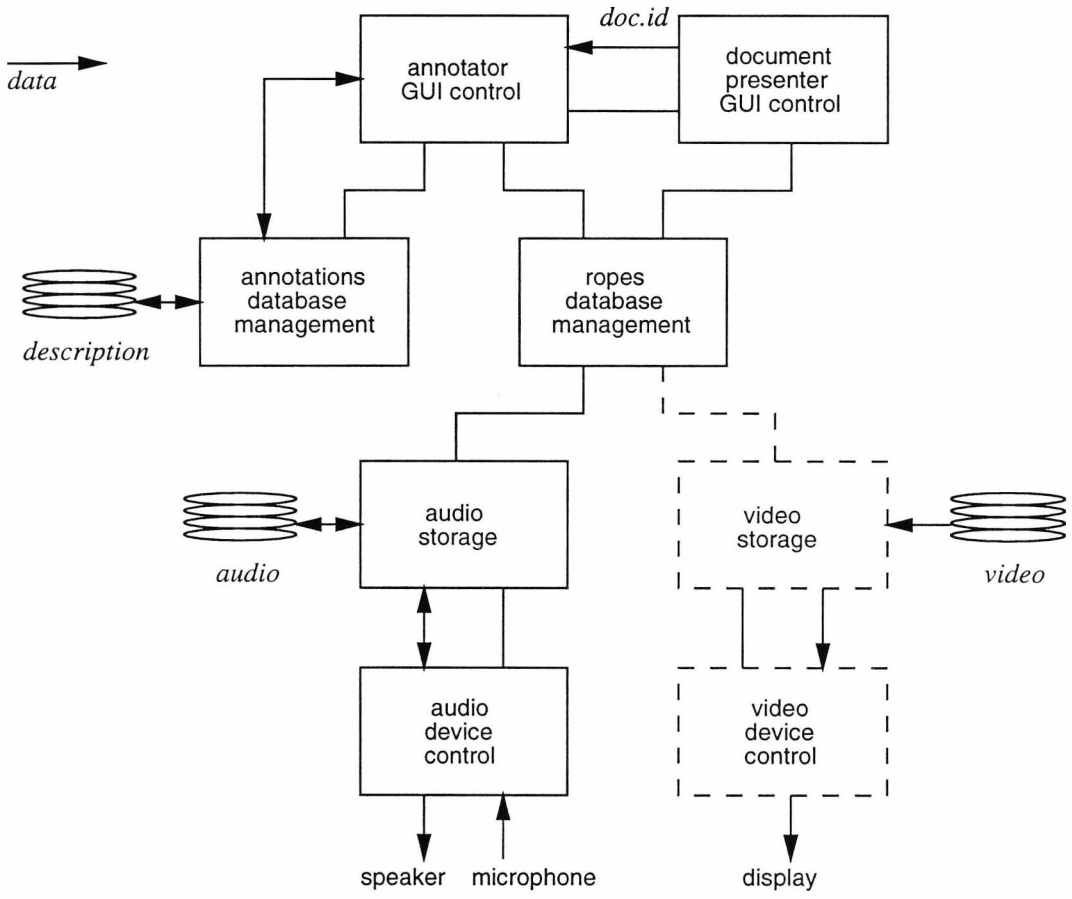
Medium device control

The necessary control of the devices that enable the presentation and capture of distinct media is included as part of the application. An audio device control module is always present to support the presentation and capture of voice for the annotations; this module is also used when base documents involve audio (e.g. music). For video documents, a video device control module is needed. These modules are linked to the respective storage modules for the storage/retrieval of audio or video.

3.2.3 Overall Structure

The application involves four basic functions: GUI control, database management, storage and medium device control. These are associated with the annotator and document presenter modules as necessary. Thus, the annotator consists of GUI control, annotations management, ropes management, voice storage and audio device control, and the presenter should follow the same model, adapted to the type of medium used (audio or video) – for audio it could use the same audio-related modules as the annotator. Figure 10 shows the structural model of the application.

The support modules are actually capable of providing services, not specifically designed for the **annotation of continuous media** application, but suitable for multimedia applications in general. The application was, in fact, planned to use existing services developed within the Networks and Distributed Systems Group at UKC's Computing Laboratory. Such services are designed to be shared by multiple clients, as in an application where the presenter is an audio-document player and may share the audio services with the annotator. The construction of the application is expected to give feedback on the suitability of the



The video–related modules are incorporated only when the document presenter is a 'video player'.

Figure 10: Application model.

audio services, particularly, with regard to aspects such as consistency and performance.

3.3 Requirements

The application requires that the media presentation runs at the established rate and that events are synchronised within minimum delays. The basic requirements are discussed as follows.

3.3.1 Bandwidth

The amount of data that needs to be sent through a given communications circuit per second depends on the media type and quality required. Thus, for continuous or isochronous media, where a minimum, constant sampling rate is required for meaningful presentation, the raw communication bandwidth requirements are indicated as follows.

Digital audio .

- telephone quality speech (mono) – 64 kbps;
- CD quality (stereo) – 1.412 Mbps;

Digital colour video .

- broadcast quality – 216 Mbps;
- high definition quality – from 1 Gbps.

The above requirements can be reduced by compression. In the application, telephone quality speech is enough for the voice annotations. However, the quality for other media involved depends on hardware facilities. For example, on SPARC stations, which standardly

provide 64 kbps for audio transmission, the quality of audio presentation, although poor, may be acceptable, depending on the purpose of the applications.

3.3.2 Synchronisation

In the application, a number of events require synchronisation. There are two ways of achieving synchronisation: (1) by making threads of execution wait for the synchronisation event to happen, or (2) by synchronising the logical clocks of the participating components when some significant event occurs. An example of the first is:

(a) the button play was pressed, so wait until the presentation starts;

the following are examples of the second:

(b) the button skip was pressed, so reset the components' clocks to a new temporal position that represents the skip, and proceed playing from the new position;

(c) while the base document is playing, time comes to a point where an existing annotation should be played, so the logical clocks are synchronised to allow the document and the annotation to play synchronously.

Table 2 lists the application's synchronisation events, which are all synchronised through method 2 described above, i.e. by updating the logical clocks according to the operation requirements. Events are triggered either by the user, or at a given time, or by end of reading rope data.

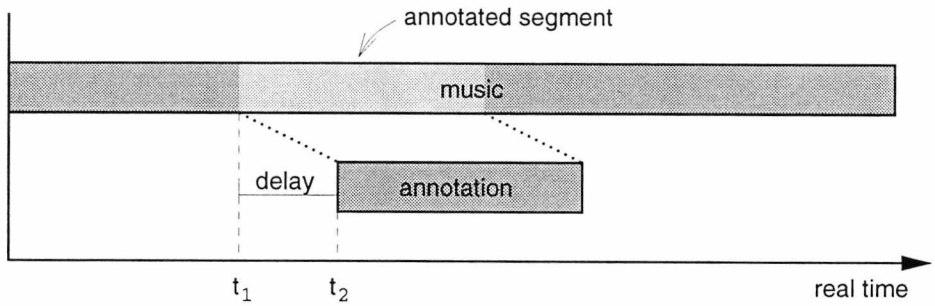
Triggering event	Main activity	Other activities
User		
• play	doc. presentation	time indicator[a]
• restart	doc. presentation	time indicator
• skip	doc. presentation	time indicator, annotation
• pause	doc. presentation	time indicator, annotation
• continue	doc. presentation	time indicator, annotation
• stop	doc. presentation	time indicator, annotation
• move[b]	doc. presentation	time indicator, annotation
• play annotation	annotation	doc. presentation, time indicator
• cancel annotation	annotation	doc. presentation, time indicator
• record annotation	annotation	doc. presentation, time indicator
Timed		
• start of existing annot.	annotation	doc. presentation, time indicator
End of reading		
• annotation	doc. presentation	time indicator
• document	time indicator	

[a] 'time indicator' displays the current time of presentation in the user interfaces (see section 4.4).
[b] The 'move' action represents an explicit manipulation of time by the user.

Table 2: Synchronisation events.

3.3.3 Real-time response

Real-time requirements aim at making sure that delays in responding to events are not so long as to degrade the presentation seriously. In the example shown in figure 11, an annotation is expected to play synchronously with a piece of music, but the longer the delay before starting the annotation, the harder the chance to achieve the requirements.



t_1 = virtual synchronisation point

t_2 = actual start of the annotation

Figure 11: Example of synchronisation in the application.

In chapter 5, response times of the implementation are measured and compared with established tolerances.

3.4 Development Environment

The application is a distributed system and needs support for interprocess communication, concurrency and synchronisation. It relies on ANSAware, introduced in section 2.4, for such support, and on X-Windows/Motif for the implementation of the graphical user interfaces. Location transparency for data files used in the application is provided by SUN's NFS (network file systems – 1.3.1). UNIX is the operating system used.

3.4.1 Distribution platform

This section reviews some facilities provided by ANSAware, as discussed in 2.4, and adds some more information about what this platform provides.

Interprocess communication

The application is designed in terms of a number of cooperating components that need to interact. ANSAware provides facilities for

- specification of interfaces that consist of the definition of types and operations for the possible interactions between components (see **IDL** in section 2.4),
- invocation of remote operations (cf. **communication**), and
- components binding (cf. **trader**).

Threads

ANSAware provides *tasks* and *threads* to support concurrency in components. A thread is a unit of potentially concurrent activity and a task is a virtual processor that supplies the resources a thread requires. When a thread executes on a task, its resources can only be freed when the thread terminates. Components can be multi-threaded, which means that they can execute more than one activity simultaneously, provided the threads have tasks to run on, because tasks are the units scheduled by the ANSA-scheduler. The use of multiple threads enables enhancements in the performance of applications, and servers to provide services to multiple clients concurrently.

Components in the application need to exhibit concurrency. For example, maintenance of the logical clock is one independent activity in some components. Clients sharing the

same service (provided by one instance of the server) also require control of concurrency. Designing the application assuming that it will be extended to support group collaboration implies that all the components require concurrency support.

Synchronisation mechanisms

The behaviour of an application is determined by the order of events it produces. When an application is composed of concurrent components, each event that actually occurs is a possible event in the independent behaviour of each separate component involved [Hoare 85]. Thus, programming constructs are needed to help implementing the desired behaviour.

ANSAware provides mechanisms for interprocess synchronisation and for inter-task synchronisation in components that exhibit concurrency. The former is achieved by the use of **interrogation operations**, in which a client waits for the results it requested, whereas the latter is provided by the facilities of **event counts** and **sequencers**, which combine to control the number of occurrences of events of given types (cf. section 2.4). The use of primitives such as **wait** and **advance** allows concurrent and distributed applications to control the relative ordering of events.

Using X-Windows from ANSAware applications

In order to prevent interference caused by certain X functions with the ANSA-scheduler, the ANSA-scheduler and the X-scheduler interact, so that the input processing of X events is handled by ANSAware, and the output by the applications. ANSAware manages the communication between clients and the X server, allowing distributed applications to access the X11 Toolkit, thus supporting window-based user interfaces.

3.4.2 Network

The application is to run on a local area network (LAN) composed of SUN SPARC 2 and SPARC 10 workstations connected via Ethernet. In theory, Ethernet's speed is defined as 10Mbps. However, according to Protogeros *et al* [Protogeros 90], studies of real Ethernet networks show that only a few of them operate near the maximum throughput, with typical loads below 50% and often close to 5%. Traffic usually consists of many minimum-length packets (64 bytes, including a 14-byte header and a 4-byte 'frame check sequence', but excluding the preamble and synchronisation bits which are stripped from the packet upon reception), some maximum-length packets (1518 bytes) and a few of intermediate size. These authors consider, therefore, that it is safe to assume a typical network load of 20% and an average packet length of 200 bytes, so that the network would carry about 4500 packets per second. This means that typical Ethernet networks' throughput is approximately 7 Mbps. Thus, the bandwidth requirement of 64 kbps for telephone quality audio can be easily achieved given such throughput.

3.4.3 Limitations

The numbers shown above, compared with the bandwidth requirements presented in section 3.3.1, demonstrate that the Ethernet is not able to carry real-time video or high quality audio, especially without compression. An experiment [Henshaw 94] in the Networks and Distributed Systems Group using an ATM network and video compression (JPEG) by hardware has shown satisfactory results.

The fact that UNIX is not a real-time operating system also presents a limitation in the sense that audio and video presentations can be disrupted by higher-priority activities.

Additionally, in a networked environment, the possibility of *jitter* must be considered. In a distributed system, a delay between a packet being sent and the same packet being received is known as end-to-end delay, and this delay may vary. Jitter is defined as the maximum difference between end-to-end delays experienced by any two consecutive packets [Zhang 91]. Quality of service in situations like the ones described in this paragraph, presentation disruption by higher-priority activities in non-real-time operating systems and the possibility of jitter in distributed systems, can be improved by careful use of buffering [Linington 90, Steinmetz 96].

3.5 Conclusions

An application capable of integrating continuous media objects has been designed, allowing users to compose voice annotations with continuous media presentations in the form of documents such as music, video clips, graphics animations, etc.

The application consists of two modules (or sub-systems), an annotator and a document player, which need support for functions such as storage, database management and device control. Storage requirements involve the storage of annotation descriptions and of continuous media, database facilities abstract the use of file operations, and device control has to do with specific types of media. The process of design involved issues like orthogonality, generality and open-endedness, and the identification of modules that display similar functionality has allowed reuse and so reduced effort. The object model provided by ANSAware, together with its support of concurrency and communication, have enabled the design of this application whose components can be distributed and synchronised. Effective synchronisation is needed to support the application's real-time requirements.

This chapter ended in a discussion of the environment in which the application will be implemented and of its limitations. The next one will concentrate on the issue of interaction, defining the application's GUIs and specifying the communication interfaces between the components.

Chapter 4

Implementation

This chapter discusses the implementation of the application designed. The relationship between the presentation of continuous media and annotations is given by the annotation structure defined. The application components' integration is discussed and their communication interfaces are specified. The presentation of the GUIs involved in the application follows the discussion of the components' interaction, and aspects of the user interaction with the application are considered. The chapter finishes by examining the implementation of the temporal access control operations (e.g. pause) of the application.

4.1 Introduction

The application designed allows continuous media documents (e.g. music, video) to be voice-annotated. The application involves two main modules: one to control the presentation of the *base document*, that is, the document which is being studied and gives sense to the annotations, and the other to support the annotation process. Support for data

storage, transfer and database organisation is required. The design has taken into account the possibility of distribution of the application's components over a network, and thus, the implementation relies on ANSAware, a distribution-support platform, to realise the application.

The application is composed of a number of objects that interact with each other based on the client/server model. ANSAware gives means for objects to define interfaces for the services they provide, and for these interfaces to be accessible to clients. This platform also supports concurrency to allow servers to handle concurrent, multiple requests, and this facility (*threads*) can also be used to enhance the performance of components of the application, and therefore, of the application as a whole, since independent activities within components can execute simultaneously. A mechanism based on event counts and sequencers permits concurrency control. ANSAware allows objects to provide multiple different interfaces and clients invoke operations in the interfaces via the RPC mechanism.

The implementation of the application on the annotation of continuous media makes use of ANSAware's facilities enabling component objects to define their communication interfaces, interact with one another, and exhibit concurrent activities such as user-input and rate control. Rate control, in particular, is a mechanism that gives application components control of a logical clock, and a common clock among communicating objects can be set, with distributed updates being done via RPC. This mechanism is used in the application to implement media synchronisation and operations that control time-dependent presentation (i.e. temporal access control operations).

4.2 Annotations

An annotation is a piece of information that refers to a part of or point in a document, commenting on it. In our application, as the base document is *continuous*, i.e., its presentation is contiguous in time, annotations refer either to instants or intervals of the document presentation. In this sense, annotations are, respectively, classified as

- **instant-annotations**: presentation of the main document is paused and the annotation proceeds, and
- **interval-annotations**: the base document and the annotation flow together.

Figure 12 illustrates the two types of annotation in a timeline representation of a session involving playing some music and associated annotations.

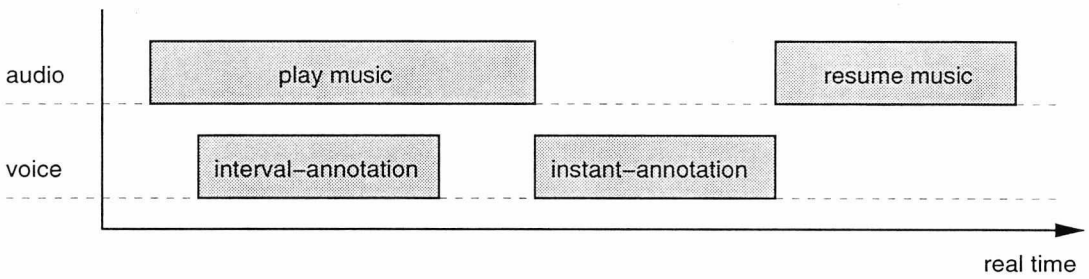


Figure 12: Timeline representation of an annotated-document.

4.2.1 Relationships between Annotations and Presentation

Annotations and the base document are related in various ways. The basic reference that links them is *time*. An annotation is a data structure that indicates

- the path of the file which stores the voice data representing a user's comment on the base document,

- the point in the document's time when the annotation should start, and
- the time when it should terminate.

Databases of annotations associated with specific documents are maintained by an annotations server, discussed later in this chapter. The type definition of an annotation is given as follows.

```
typedef struct _annotation_type {
        char *filepath;
        int init_time;
        int end_time;
} annotation_type;
```

In order to be meaningful, annotations are expected to play at normal speed (100% – factor 1), so their duration is derived from the respective voice file's length. However, the user may wish to comment on the document playing at a different speed (e.g. half or double) or at a different direction (e.g. backwards) – the sign of the speed factor can, implicitly, express the direction, and the value 0 (zero) can represent pause. Therefore, the ratio of the difference between the end time and the initial time to the annotation duration determines the speed factor of the document's presentation during the occurrence of annotation.

$$speed\_factor = \frac{end\_time - initial\_time}{annotation\_duration}$$

Table 3 shows examples of the relationships between annotations and document presentation.

The established relationships require temporal transformations [Little 94] to be made in the document presentation when annotations occur, as seen in figure 13.

Initial time	End time	Duration	Ratio	Presentation
10	20	10	1	normal speed
50	50	15	0	paused
60	70	20	1/2	half speed
100	120	10	2	double speed
150	140	10	-1	inverted

times in seconds.

Table 3: Examples of presentation during annotation.

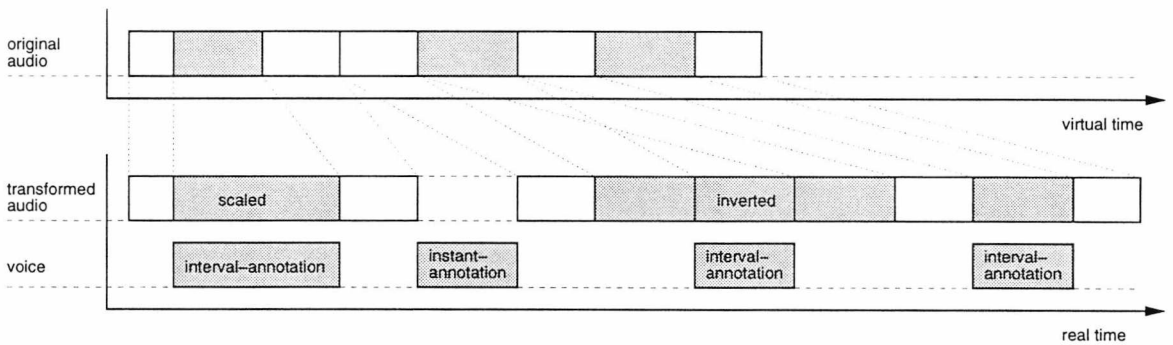


Figure 13: Presentation transformations determined by annotations.

Observe that the interval of the document whose associated annotation requires *inverted* presentation is played three times:

1. play until initial time of the annotation is reached;

2. play backwards to conform with the annotation requirement;

3. finished the annotation, play forwards.

The application recognises such situations to prevent infinite loops. The necessary constraints are discussed in section 4.5.

4.3 Component Integration

The application was designed as a collection of components that interact providing services to one another. In this section, the requirements for the integration of the components are discussed and the configuration of the application is presented.

4.3.1 Requirements

In a distributed application, requirements involve establishment of connections among the components, control of synchronisation and data transfer.

Binding

Interfaces describe the service operations provided by servers. Possession of an interface reference by a client allows it to invoke operations at that interface (see section 2.4). References can either be obtained via the trader or be passed from one object to another as arguments or results of operation invocations.

Services offered (*exported*) to the trader are made public, and as such, they may be used by different client applications. (The trader organises the offers it receives according to *type*, *context*[1] and (optional) *properties* (e.g. aliases, user names).) On the other hand, interface references passed within a particular application represent internal services. In the following, the exported and the non-traded interfaces of the annotations application are listed.

- Exported interfaces: a client wishing to use a certain service has to provide appropriate parameters about it to the trader, so that a service instance can be searched for and the

[1] The context-space in ANSAware is organised as a hierarchical tree.

client can obtain an interface reference for it. In the application, the public interfaces (types) that can be imported are:

  - `docPlayer`: this interface is offered by the document player to indicate that it wishes to integrate with another object; it is identified via a user login name, so as to ensure that the annotator, for example, binds to the player used by the same user;
  - `ads`: the annotation database server's interface; it may be identified via an alias (e.g. "AnnotDB");
  - `vrs`: this interface is offered by the voice rope server, which controls the rope database; it can be identified through an alias (e.g. "VrsAnnotator");
  - `ass`: `ass` is the interface for the audio storage service; similarly, an alias (e.g. "AssAnnotator") can be used;
  - `audctrl`: it represents the connection establishment service, provided by the audio server; usually, user login names identify this interface.

- Non-traded interfaces: these are the interfaces whose references are passed within the application:
  - `pPartner`: the interface for the presentation partner (e.g. the annotator);
  - `rate`: the rate control interface;
  - `callback`: the interface used to report back to a client the status of audio operations (e.g. play started);
  - `audio`: the interface used for audio data transfer.

The interfaces are described in section 4.4.

Rate communities

Objects involved with the same continuous medium document must perform rate control synchronously, and so form a rate community. In the application, the objects are those that control the user interfaces (music player and annotator), manage the voice and music rope databases (rope server) and control data transfer (audio storage server). Synchronisation is achieved via the `rate` interfaces, which are provided by the objects participating in the same community – the ropes server is also the server of the rate communities. Given that the various types of relationship between annotations and base document are, basically, defined by how they are rate-controlled, separate rate communities are created, and the management of the relationships is done by the annotator. Objects provide as many `rate` interfaces as the number of different communities they are engaged in.

Audio transfer

Audio data transfer is made within the audio sub-system between the audio storage server and the audio server. Since the audio server encapsulates the hardware to make sound in the workstations, one instance of the server must run on each station. Therefore, the audio storage server has to know the correct reference of the `audio` interface with which it should interact. That is quoted by application clients when they request transfer to occur in connection with specific stations. For example, when the music player requests for a rope to be played on the workstation A, it specifies the `audio` interface reference related to the server that is running on that station. That reference is obtained via the `audctrl` interface (cf. section 4.4.5). When the audio server is started on a workstation, it exports the audio control interface with properties that, implicitly, associate it with the station. The application then has to make sure that clients import that interface with the right properties.

4.3.2 Configuration

The connection of objects to one another in the application is shown in figure 14. Connections are made regarding

- **audio**: involving the audio server, the audio storage server (assServer), the rope server (vrsServer), with the annotator and the music player as clients. The clients request the audio server to create the audio interfaces (channels) for voice and music. The references for these interfaces are quoted by the respective clients to the rope server, which passes them to the storage server so that voice and music data can be transferred to/from the appropriate channels. Additionally, the annotator and the music player create their callback interfaces, enabling them to receive reports from the storage server;

- **rate control**: this involves music presentation and the voice annotations. The music rate community is formed by the music player, the rope server, the audio storage server, the audio server, and the annotator; and the annotations rate community consists of the annotator, the rope server, the audio storage server. As mentioned earlier, the annotator is responsible for avoiding conflicts between annotations and the underlying music in terms of their presentation rates;

- **annotation description storage**: the annotator communicates with the annotations database server (AnnotServer) via the ads interface;

- **sub-systems interaction**: the annotator registers its callback interface pPartner (presentation partner) with the music player via the docPlayer interface. The two sub-systems can interact through these interfaces.

The interfaces mentioned here are described in the following section.

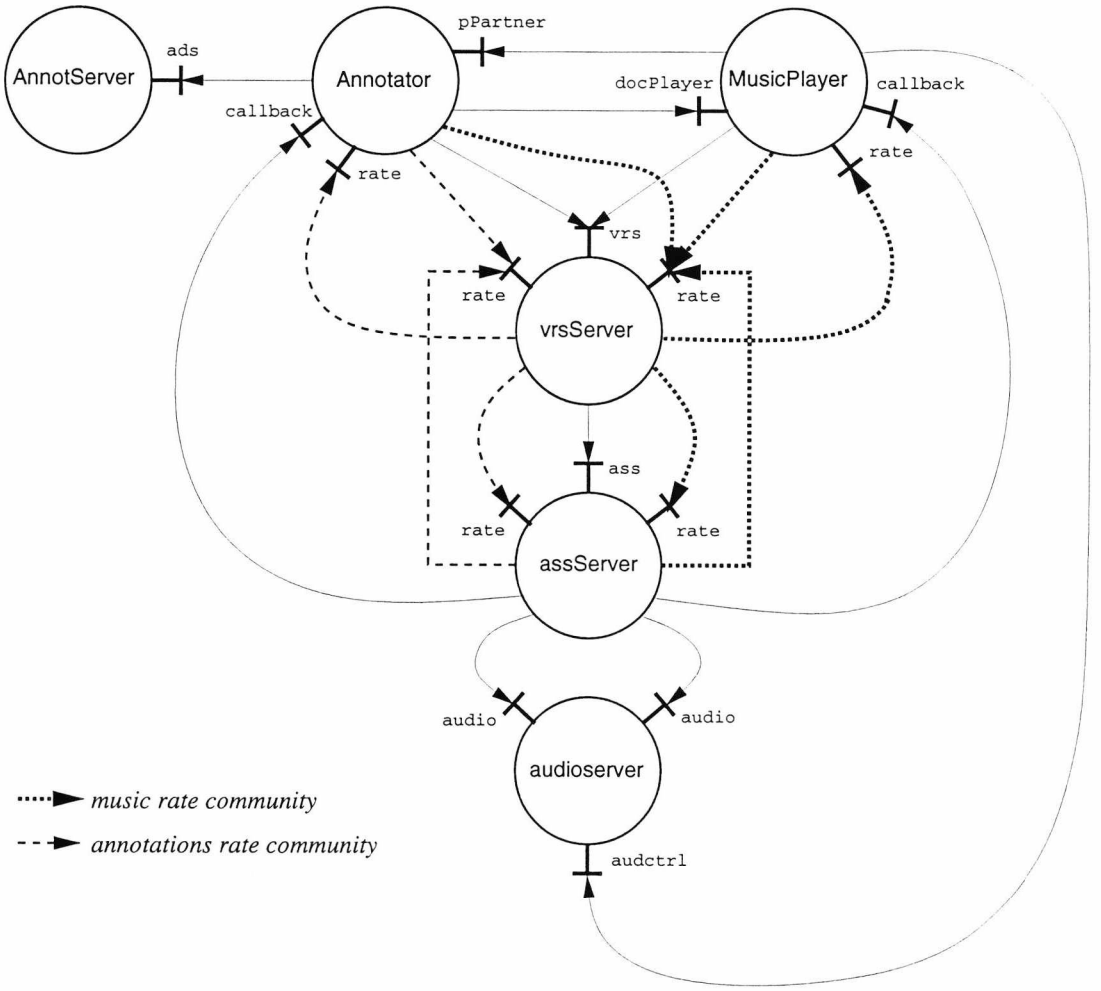


Figure 14: Configuration of the application.

4.4 Communication Interfaces

Services are provided by objects that represent the components of the application. Objects interact by calling operations on interfaces that present an abstract view of the services. This section discusses the services and presents the operations used in the application (the

interfaces' IDL descriptions are given in Appendix A). Many of the interfaces shown here have not been designed to be used exclusively in this application, so reusability can be explored.

4.4.1 Rate Control

Each object in the application responsible for a continuous media document (e.g., voice and music) needs to control the rate of presentation. An interface is defined to allow synchronisation among the components that support it. In order to establish a *rate community* one of the components is defined to be the *rate server.*

The `rate` interface

This interface allows control of speed and direction of time-dependent media, defining a logical clock with respect to real time. For each setting operation, the result is the value actually set, which may not be the same as the request if the device cannot support all options.

Speed. Speed is specified as the ratio of logical time to real time (logical/real ticks). Thus (1,2) is understood as *play at 1/2 speed.*

Direction. Forward (true) or backward (false) direction can be set directly through this operation, or implicitly by negative speed.

Pause. This operation allows temporary pause (true) without change of speed or direction. Resumption is accomplished by setting pause false.

Position. Positioning in the time-dimension specified by the associated length is made via this operation.

Update. This operation is used to update, in one call, the presentation variables speed, direction, pause and position.

Request. This operation allows the current values of the presentation variables to be requested from time to time, enabling synchronisation among the components of the same rate community.

In addition to the above operations, the management of the rate community by the server is supported by

Register. This operation is used by a client which is interested in the service. A callback interface reference of type `rate` is passed as argument, and in return the server sends a status flag and a handle by which the client is known; and

Deregister. Which is called when the client is no longer interested in the service.

4.4.2 Music Player

The object responsible for the presentation of the base document (e.g. music, video) expresses its desire to integrate with other objects (e.g. the annotator) by offering the following interface.

The `docPlayer` interface

This interface provides the following operations.

Register. The interface reference for the object that registers interest in integration is received. It is associated with a handle, which is sent back together with the status of the operation. Also, the interface reference of the *rate server* is returned to enable the client to register itself with the community including the document player, so that synchronisation can be achieved.

Deregister. Integration is discontinued via this operation. The handle indicates which object is being de-registered, and an acknowledgement is produced.

4.4.3 Annotator

Any object that is to be integrated with the document player, whose interface has been described above, must produce the "presentation partner" interface.

The `pPartner` interface

This interface involves the operations:

Selected Document. Through this operation the presentation partner (i.e. the annotator) is informed that a document has been selected for presentation. The name of the document is specified (enabling annotation naming) and an acknowledgement is returned; and

End Session. This is used to request the end of session. The operation is acknowledged.

4.4.4 Annotations Server

The annotations database server is responsible for maintaining the consistency of information about annotations. Its service is provided by the following interface.

The `ads` interface

This interface has been designed to allow the application to develop from single to multiple user support. If there are multiple users, clients have to provide a callback interface, described in Appendix A. (Group collaboration issues are discussed in Chapter 7, under proposals for future work.) The interface operations are

Register Document. A document handle is produced in return to the given document name.

List Annotations. Using the document handle, the client can request the descriptions of associated existing annotations[2].

Store Annotation. This operation is used to store information about a new annotation associated with a document handle. The result of the operation is the annotation name given by the server.

Deregister. Via this operation a client informs that it will no longer use the service.

4.4.5 Audio Server

The audio server encapsulates the hardware, providing audio support based on the facilities available on SUN SPARCstations. This object produces two service interfaces to clients.

The `audctrl` interface

This is the main interface of communication between clients and the audio server. It is used for connection establishment and provides three operations. One creates a port, and

[2]The voice-related data are managed and stored by the voice ropes (section 4.4.6) and audio storage (section 4.4.7) servers, respectively.

the other two are for indirect connection. These are not used in our application, so refer to [Linington 90, Linington 91] for appropriate explanations. Here, only the first operation is described.

Create. This operation returns an interface reference to be used for audio transfer. If the interface cannot be created, an error indication is returned. The transfer mode required (simplex or duplex) and the user identity are given as parameters.

The `audio` interface

This interface allows the exchange of audio in a series of spurts between client and server objects. Service can only be provided to clients that possess a reference for the interface. These are not exported to the trader; they are obtained via the control interface and may be passed around in the application. Only one operation is supported by the audio interface:

Spurt. Playback, recording or both are supported by this operation, depending on the mode established for the interface when it is created. Spurts are transferred as sequences of PCM speech samples. The delay expected to happen before the spurt begins to be played is reported through this operation, which also carries status flags for normal or end-of-stream actions.

Currently, 512 bytes of audio data may be transferred in each direction by each RPC; serialisation of RPCs provide flow control. Buffering is used to prevent disruption. (These facts are considered in the performance evaluation of the application, discussed in Chapter 5.)

4.4.6 Rope Server

This object supports an abstraction called a *voice rope*, which represents a sequence of stored voice, so that clients need only to refer to ropes when they want to record or play voice. It manages the data structures that represent the abstraction and supports sharing among clients. Originally described in [Li 92], the server has evolved to provide the interface defined in Appendix A, and discussed as follows.

The `vrs` interface

Here, only the operations used in the application are considered.

Request Rate Interface. It is natural for the rope manager to be the rate server. Thus, this operation is used by a client to request the establishment of a new rate community. The results are the interface reference for the rate server so that clients can register interest to synchronise, and the status of the operation.

Register Rope. Register the details associated with a rope. In return, the client knows the rope-handle and the rope's length.

Play Rope. A registered rope is played through this operation.

Cancel Play. Stop rope being played.

File to Rope. Create a rope from a given file.

Register Record Rope. This operation allows the recording-specific details of a rope to be registered.

Record. Request for recording a rope is made via this operation.

Cancel Record. This operation stops the recording of a rope.

4.4.7 Audio Storage Server

This object provides only one interface, which is fully described in Appendix A. Here only the operations used in the application are considered.

The `ass` interface

New Rate Handle. Through this operation, the audio storage server is required to create a rate interface and to join a community by registering with the given server interface reference. The results are the status of the operation and a handle corresponding to the client rate interface.

Register Rope. The necessary information to enable the server to read audio data referring to a rope is given via this operation, which returns the associated rope handle for future reference.

Play Rope. This operation enables the storage server to start reading data associated with a certain rope.

Cancel Read. Used to cancel data reading.

Write. The server is requested to store incoming audio data through this operation.

Cancel Write. The server is requested to stop writing.

Application clients may wish to be informed of certain events that occur during audio transfer. For this, they have to provide the callback interface described below.

The `callback` interface

This event callback interface supports the following operations:

Failed. Used to report processing failures in the audio store.

Cancelled. This operation is used to indicate that the reading/writing process has been terminated.

Done. Called when rope playing/recording is finished.

Event. Reports the occurrence of user-defined labels in the rope.

Paused. This indicates a pause in the playback of a rope.

Started. Called when a rope has started to play.

4.5 Interaction

Here are presented the layouts of the graphical user interfaces (GUIs) of the Music Player and the Annotator. The use of their features and their combined reactions to user requests are explained.

4.5.1 The MusicPlayer User Interface

This interface provides the basic features found in any continuous-medium player, like a VCR or a CD-player. It works as a remote control that gives the user the ability to

- *play* or *restart* from the beginning,
- *skip* either forwards or backwards,

- *pause* the presentation,
- *continue* from a paused situation,
- *stop* the presentation,
- move across it, and
- change speed and direction.

Additionally, the interface provides

- selection of the document to be presented and
- a time display.

Figure 15 shows the MusicPlayer user interface, which consists of three areas:

Timeline. This area's length denotes the total duration of the presentation and a slider in it shows the current time. This slider can be dragged, representing time manipulation.

Speed/direction setting. Speed and direction of the presentation can be changed through the slider in the scale. The centre of the scale indicates zero speed, and to the left of it negative speed is set, whereas positive speed is set by moving the slider to the right of the centre of the scale. Default values are 1 (normal speed) and true (*forwards*), respectively.

Buttons. Each button in this area activates a specific operation that affects the presentation – the complementary operations *pause* and *continue* share the same button. The button select, which allows selection of the document to be presented, cannot be used during a presentation.

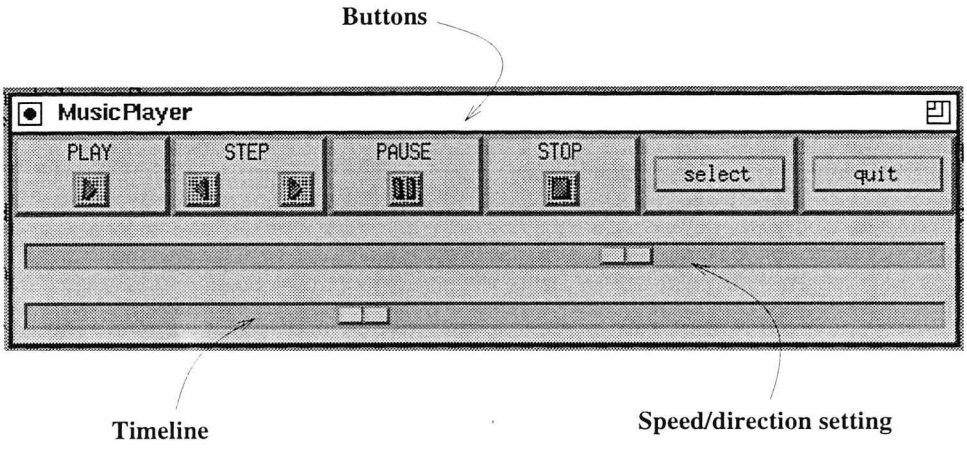


Figure 15: The MusicPlayer GUI.

Constraints

The following interaction constraints are given for the system's operation.

(a) Document selection can only be made in the idle state.

(b) Skip cannot be used to start playback.

(c) During pause, only the button continue can be pressed, de-activating the state. In this state, the timeline slider cannot be dragged.

(d) Speed and direction can be set at any time, including pause, without modifying the current state.

The complete set of state changes can be seen in figure 16, where idle is the initial state and finished is the final one.

4.5.2 The Annotator User Interface

The Annotator interface can

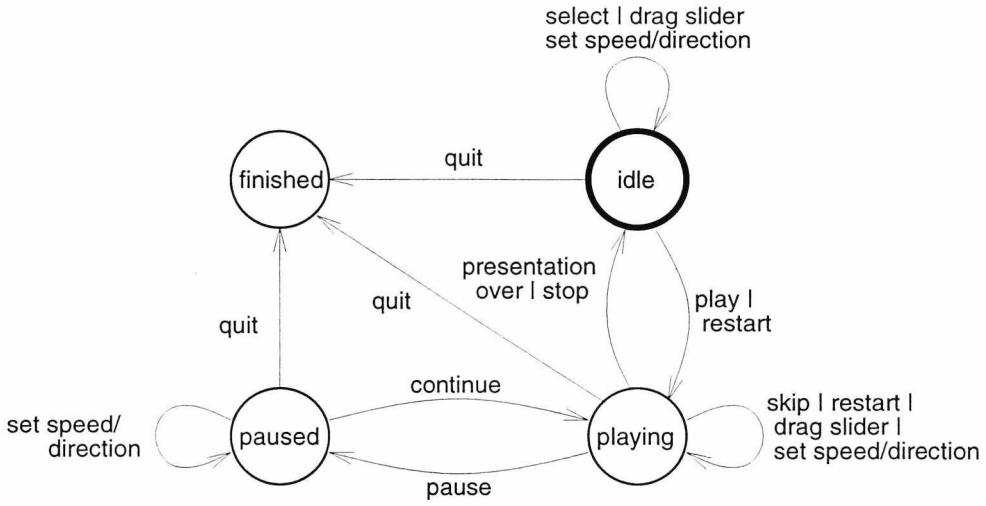


Figure 16: MusicPlayer states machine.

- *denote the total duration of the main document's presentation.* The length of the timeline designates the total duration of the presentation and works as a time-scale;
- *show the current time of the presentation.* As the presentation progresses, a slider flows in the scale to represent the time passing;
- *indicate where annotations are found, relatively to the presentation time.* Since the total duration of the presentation is represented, the positions of annotations are graphically displayed in the interface, showing the correspondence between the comments and the points in the presentation to which they refer;
- *give the user control of the presentation.* The slider allows the user to move across the presentation, viewing or reviewing it from a new location – synchronisation between the annotator and the music player is necessary; and
- *permit voice annotations to be recorded and played back.*

The GUI given in figure 17 presents three interaction areas, which are described as follows.

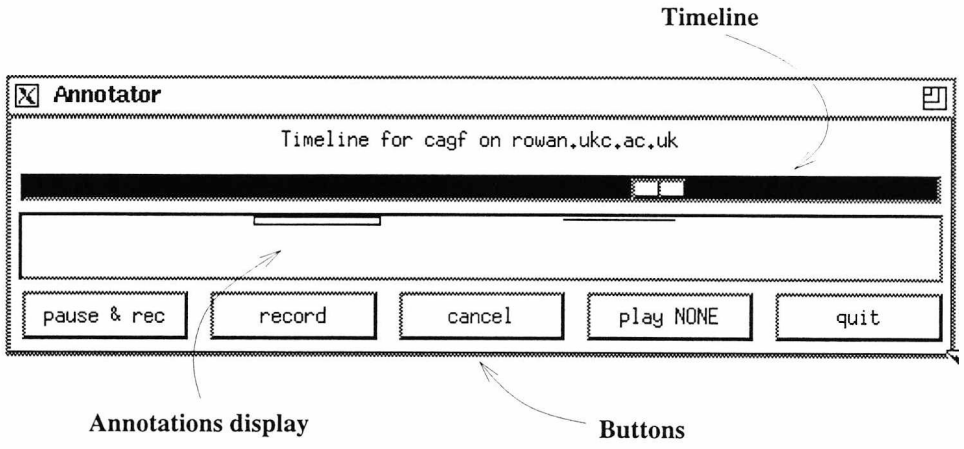


Figure 17: The Annotator GUI.

Timeline. This area enables the representation of the duration and current time of the presentation. The slider can be dragged for the purpose of time manipulation.

Annotations display. Here the annotations are mapped onto space. Random access of annotations displayed in this region allows them to be played back – this is an implicit form of time manipulation.

Buttons. This area consists of the following buttons:

- pause & rec, to permit the recording of **instant-annotations**,
- record, for **interval-annotations**,
- cancel, allowing the cancellation of annotation recording and playback,
- play-status selection, through which the user can decide whether to play NONE or play ALL or play SELECTively the annotations associated with the presentation – the label shown on the button indicates the current play-status, and

- quit, to disconnect the Annotator from the MusicPlayer.

Interaction rules

The following rules are imposed.

(a) Play-status selection can be made at any time, but if there is an annotation being played or recorded, this is not affected. The meanings of the play-states are described as follows.

- NONE. No annotation can be played either through selection or when the presentation's current time corresponds to an annotation's initial-time.
- SELECT. Annotations can only be played through user selection.
- ALL. Selection and current time may activate annotation playback.

(b) No kind of rate manipulation can be made while an annotation is being recorded. Only the cancel operation is allowed in order to finish the recording, retrieving the previous state.

(c) During annotation playback, a new annotation can be made. However, limitation on the number of simultaneous annotations is necessary to ensure clarity, and also because there might not be enough memory available for the tasks needed to accommodate all the threads that would have to run concurrently, forcing threads to queue for a task to become available.

(d) Selection of an annotation when another is being played implicitly cancels the current playback. Nevertheless, more than one annotation can play concurrently provided they are activated by time and the limit of simultaneous annotations is not exceeded.

(e) Dragging the slider in the timeline does not initiate annotation playback if initial times are reached, but annotations which are already playing behave accordingly.

The possible actions in the different states of the Annotator sub-system are shown in figure 18, in which idle is the initial state and finished the final one.

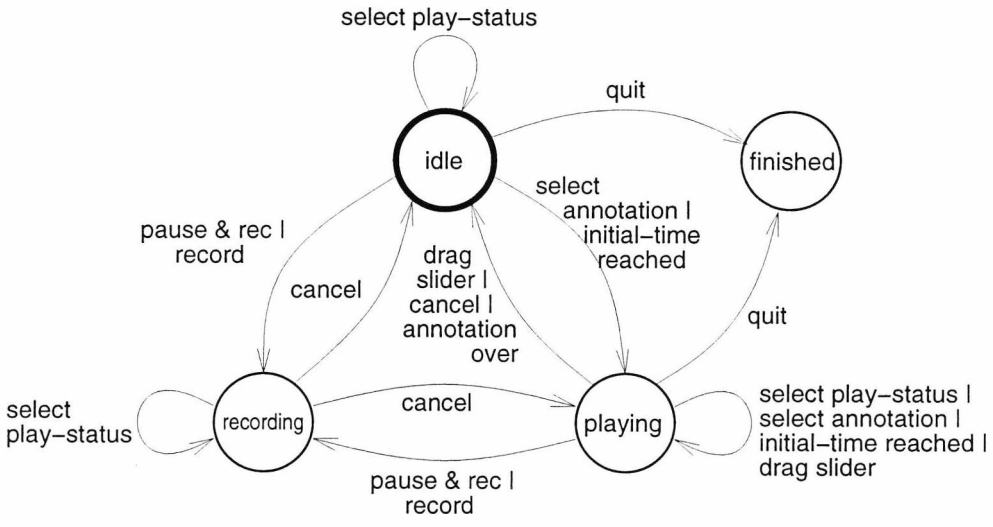


Figure 18: Annotator states machine.

Graphical resolution

The timeline should be able to represent the duration of a continuous medium document by mapping time to space, and according to Myers [Myers 85], a graphical interface should give a clear view of position in the time-space, so that the user can navigate more confidently through the presentation. Given the variety of durations, the continuous position updating of the slider uses more or less space in the timeline. Lengthy documents' presentations, in particular, use much less space per unit-time, with the slider's motion appearing too slow, and so, dragging it in the timeline may deliver undesirable long jumps in the presentation,

making the navigation difficult. Moreover, displaying annotations, as in the timeline of the annotator, in cases when the timeline represents lengthy documents, can generate misunderstandings about the nature of the annotations (whether they are instant- or interval-annotations) because little space is used. It is not practical to vary the interface's length to solve the resolution problems of time updating and duration indication (for base document and annotations) because of the diversity of duration of continuous media documents (music, movies, etc.). In MIT's MegaSound [Hindus 93], the interface can incorporate an additional level to expand pieces of the timeline without losing global positioning information. User interaction can be with either the original timeline or the zoomed one, as their behaviours are similar. This is a possible solution that could be used in the interfaces of the annotator and the document player.

4.5.3 Common Interaction

The MusicPlayer and the Annotator have their own functions, but cooperation and interaction give them the ability to deliver combined reactions to user requests. Actions taken in one interface may modify the behaviour of the other sub-system, and vice-versa, so a protocol has to be defined to make sure that user's expectations are fulfilled.

Annotator's interference on the main document presentation

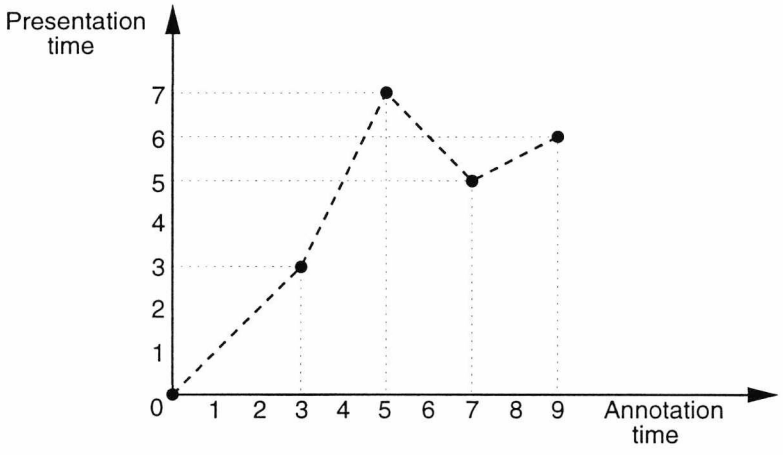
Besides the operations that can be invoked during the music presentation through the MusicPlayer interface (skip, pause, etc.), others can be made via the Annotator:

- *time manipulation*: moving the slider in the timeline makes the presentation behave accordingly, i.e. play backwards or forwards at different speeds;

- *annotation playing*: as time reaches the initial-time value of an annotation, this starts playing synchronously with the presentation. Optionally, the user can select an annotation in the annotation display, updating the current time of the presentation to be the initial time of the selected annotation, providing random access in the presentation. In any case, if the annotation is an instant-annotation, the presentation pauses during the occurrence of annotation;
- *annotation recording*: this can be made during the presentation or by pausing it.

Effects of presentation manipulation on annotations

During annotation playback, any sort of rate-manipulation operation can be invoked, requiring corresponding behaviour. Some operations may have the effect of cancelling the playback, like skipping to a position in the presentation outside the annotation's time-interval. Annotation recording is treated specially to prevent bizarre effects: although annotations may refer to the presentation at different speed or direction, these have to be set previous to the recording and remain constant throughout – an annotation's recording speed and direction are **always** normal and forwards, respectively; during an instant-annotation recording, however, since the presentation is paused, speed/direction setting for the presentation can be made as its effects only appear when the recording finishes. The restriction of keeping speed/direction constant during annotation recording simplifies the representation of the relationship between document presentation and annotation, both in terms of the annotation structure (cf. section 4.2.1), and graphically in the Annotator's GUI. Similarly, no other rate-manipulation action (e.g. restart) can take place while an annotation is being recorded. In figure 19, a scenario involving changes during annotation recording shows some of the information that would need to be stored if that restriction were not imposed.



+ At (0,0) start recording, normal speed, forward direction
+ At (3,3) double presentation speed, forward direction
+ At (5,7) normal speed, backward direction
+ At (7,5) half speed, forward direction
+ At (9,6) end recording

Figure 19: An example of how rate might be manipulated during annotation if doing so was allowed.

Conflict handling

Conflicts, such as the simultaneous occurrence of two annotations that require different speeds or directions for the presentation of the base document, are administered by the Annotator, giving priority to the annotation that started first. Since the Annotator allows only one annotation to be recorded at a time, conflict can exist between playing multiple annotations or between recording a new annotation and playing previous ones, but never between the recording of two annotations. Annotation recording always gets priority. So, in general, only the activities with priority can proceed, and simultaneous occurrence of annotations can only happen if they all have the same requirements for speed and direction for the presentation of the base document (see examples in figure 20). In practice, such a priority policy means that some annotations can only be played through user selection.

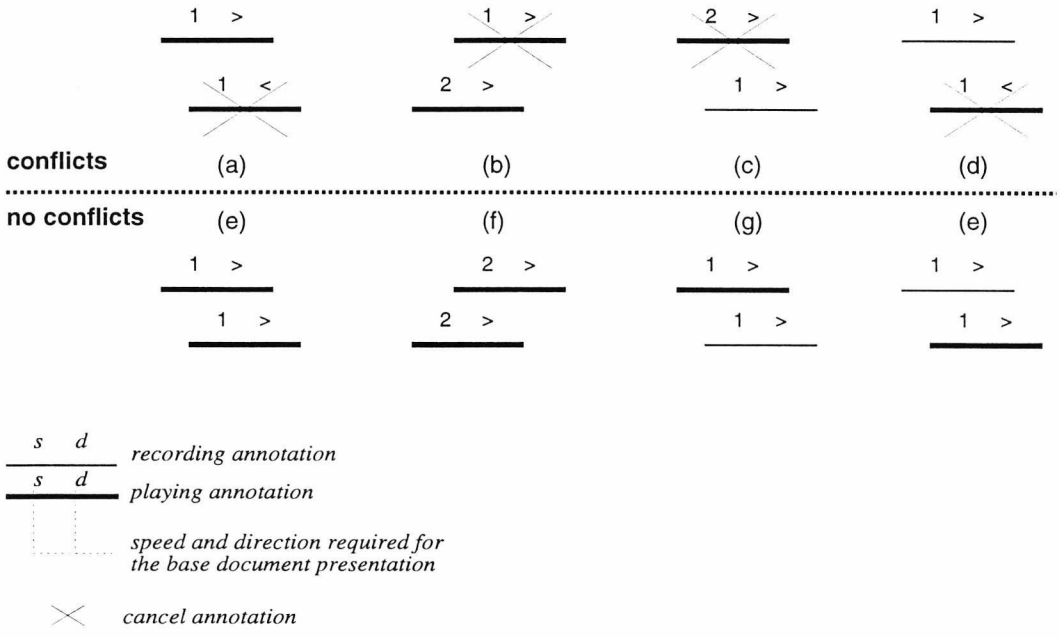


Figure 20: Examples of conflicting and non-conflicting annotations.

Communication protocol

To ensure a smooth interaction between themselves, the MusicPlayer and the Annotator negotiate a "global" status, both using the interface below.

```
CommonIntf : INTERFACE =
BEGIN

-- List of the possible common states negotiated between
-- the Annotator and the MusicPlayer

   GlobalStatus : TYPE =
      {Idle,
       -- (0)
       Playing,
       -- (1) main document presentation only
       Paused,
       -- (2) presentation paused, no annotation
```

```
        SyncPlay,
        -- (3) annotation and presentation synchronous playing
        SyncRec,
        -- (4) annotation recording synchronised with presentation
        AnnotPlay,
        -- (5) annotation playback in paused presentation
        AnnotRec
        -- (6) annotation recording during paused presentation
      };

-- The following operation allows status negotiation

   NewStatus: OPERATION [ submitted_status: GlobalStatus ]
              RETURNS   [ new_status: GlobalStatus ];

END.
```

The states defined in the common interface – `0` or `Idle`, `1` or `Playing`, `2` or `Paused`, etc. – are considered in the following, where the *global policy of interaction* is described. The table represents either states in which each action cannot be taken (–) or states resulting from possible actions taken in each possible state – when more than one state can result from a particular action in a certain state, the resulting states appear separated by | (representing *choice*).

	States						
Actions	(0)	(1)	(2)	(3)	(4)	(5)	(6)
select document	0	–	–	–	–	–	–
set speed/direction	0	1	2	3	–	5	6
play/restart	1 | 5	1 | 5	–	1 | 5	–	–	–
stop	–	0	–	0	–	0	–
skip	–	1	–	1 | 3	–	–	–
pause	–	2	–	2	–	–	–
continue	–	–	1 | 3	–	–	–	–
presentation over	–	0	–	0	0	–	–
quit MusicPlayer	0	0	0	0	–	0	–
select play-status	0	1	2	3	4	5	6
drag slider	–	1	–	1 | 3	–	–	–
pause & rec	6(a)	6	6	6(d)	(b)	6	(b)
record	6(a)	4	6	4(e)	(b)	6(e)	(b)
select annotation	–	3 | 5	–	3 | 5(c)	–	3 | 5(c)	–
initial time reached	–	3 | 5	–	3(e) | 5(d)	–	–	–
annotation over	–	–	–	1	–	1	–
cancel	–	–	–	1(c)	1 | 3(f)	1	1 | 2 | 3(f)
quit Annotator	0	1	2	1	–	1 | 2(f)	–

Observations:

(a) provided there is a selected document;

(b) only one annotation can be recorded at a time by one Annotator;

(c) cancel current playing annotation(s);

(d) pause current playing annotation(s);

(e) do not exceed limit of simultaneous annotations;

(f) return previous status.

4.6 Implementation of Operations

The so-called temporal access control (TAC) operations [Little 94] are considered here. These can be achieved in many ways. For example, video fast-forward can be provided either by skipping frames or by doubling the rate of playout. In the following, the implementation of the various operations in the application is discussed, remembering that all the objects engaged in a same rate community should react accordingly.

- *Play/Restart.* These operations are implemented by calling `VrsPlayRope` and making `rate_Position(0)` to ensure that the rope is played from the beginning. However, to restart while the rope is being played simply do `rate_Position(0)`.

- *Skip.* This operation was designed to make the presentation skip the equivalent of 10 seconds, either backwards or forwards. That is implemented using the operation `rate_Position`.

- *Pause.* `Rate_Pause(true)` implements this operation.

- *Continue.* Midpoint resumption is achieved by doing `rate_Pause(false)`.

- *Stop*. This operation is provided by calling `VrsCancelPlay`.
- *Browsing* (slider dragging). This is achieved by manipulating rate position.
- *Annotation selection* (random access). Selecting an annotation for presentation implies that the associated rope is played, which is done by calling `VrsPlayRope`. The rate controls for the annotations and the base document are separated. Thus, while the rate position for an annotation is set to 0 (zero) to make sure it is played from the beginning, the document's position is updated to the initial time of the annotation to achieve synchronisation (cf. section 4.2.1). The Annotator manages the relationship between the base document and annotations by making appropriate use of the operations `Speed`, `Direction`, `Pause` and `Position` on the rate interface.

4.7 Conclusions

The implementation of the application on the annotation of continuous media (music has been considered) has focused on interaction. User interfaces have been presented and delivery of combined reactions between the base-document presenter and the annotator has been examined. The major issue, however, has been the interaction among the components of the application. This has been possible by the definition of interfaces encapsulating operations (and data) and associating them with objects that implement the operations. 'Client objects' can invoke operations on interfaces provided by 'server objects'. ANSAware supports object binding and communication, and a mechanism developed to support rate control has been used to enable synchronisation between the objects that participate in the presentation

of continuous media.

The use of services not especially designed for the purpose of the application has been important to check their consistency. They have proved themselves reusable in general; in fact, the building of the application has benefited from the reuse of code and objects. Nevertheless, the implementation has contributed to the improvement of some of those services. In the next chapter, the performance of some of the temporal access control operations, whose implementation has been discussed here, is measured to see whether further improvements in the implementation are necessary.

Chapter 5

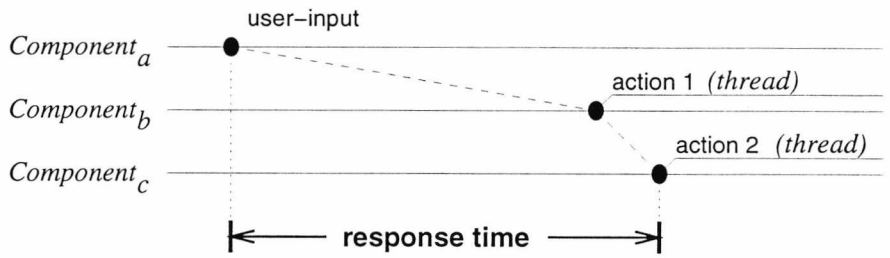
Performance of the Implementation

The performance of the application in response to a specific set of operations is examined. Response times are measured to represent the performance evolution in the different stages of the application's development. The test environment is a local network of SUN SPARCstations 2 and 10, with all the data files residing on the same disk. File access and network transmission times are considered in the measurements, and the results are presented specifying the factors that mostly affected the response times for the various operations.

5.1 Introduction

To evaluate the application's performance, response times to user actions were measured. This chapter identifies the factors that affect such response times and discusses the improvement of the performance resulting from the measurements. File access, CPU and memory use and network transmission are some of these factors. These are considered in the measurements, which take into account the processing power of machines in the

test environment. The application's response to the user events selected for observation (cf. section 5.2) involves actions performed in components distributed across the network. Figure 21 illustrates the application's notion of *response time*, which considers the time-interval that includes the user's input and the beginning of the last action taken in the application to respond the user-request. This is in accordance with the approach described in [Shneiderman 92] that considers the initial time of the responding action, instead of the end-time. That is necessary for continuous media.



$Component_a$ controls the GUI that accepts user inputs.

Figure 21: Response time in the application.

One can consider delays in two distinguished ways: (a) an acceptable delay, so that one can work with it, and (b) a delay that if it is noticed, it can make the work difficult. Some considerations that relate to our work include:

- *confirmation of physical actions*: in a classification of response times requirements (in [Coats 87]), the interval 0.1-0.2 second is recommended as acceptable, considering that the user should expect an *almost instantaneous* feedback that the action has taken place;

- *human perception threshold time*: Nicolau [Nicolau 90] talks about *direct manipulation interfaces* [Schneiderman 83], characterised by concurrent input and the provision of timely positive feedback in response to user actions, and says the feedback provided should appear instantaneous to the user, meaning that a maximum response time should be on the order of 10-40 milliseconds, i.e. human perception threshold time;

- *maximum acceptable delays for various media*: Little [Little 94] cites [Hehmann 90] for, among others, an acceptable delay of 0.25 second for voice and for video. This can be the case, for example, in the context of satellite transmission;

- *human perception of jitter and media synchronisation*: table 4 derives from results of experiments presented in [Steinmetz 96].

Media		Mode, Application	QoS
video	audio	lip synchronisation	+/- 80 ms
audio	animation	event correlation (e.g. dancing)	+/- 80 ms
	audio	loosely coupled (e.g. background music)	+/- 500 ms

Table 4: Quality of Service for continuous media synchronisation.

The experiments shown in this chapter concentrate on response times to user requests in an audio presentation. Our experience with the application indicates that an acceptable delay should be in the interval 50-80 milliseconds.

The feedback provided by the implemented application involves (a) a slider, whose position in the timeline of a graphical user interface (GUI) represents the time passing, and (b) the presentation of a music. The experiments observe such graphical and audio feedbacks by measuring the initial times corresponding to the slider positioning and the audio playback, respectively.

Two types of experimental results are shown. First, there are the results which will support the understanding of the application's performance. These include time measurements of file manipulation, involving opening and reading, and remote procedure calls (RPC) regarding data transfer. Variation of transfer size is studied to indicate its effect on file access times, as well as on communication times. Then, the results of measurements of response times to the events identified are presented to characterise the actual performance of the application, according to its development stages.

5.2 Observable Events

Time is measured referring to the

- *operation request*: a button click by the user,
- *slider positioning*: system's update of the slider's position and state (paused or continuing) in the timeline, and
- *audio response*: halt, or play from a specified *position* in the playback time.

Figure 22 shows the achievement of the skip operation, in which the user expects to view a jump of the slider in the timeline, equivalent to 10 seconds, and to listen to the audio accordingly. Notice the use of the rate mechanism, controlled by the rope server (vrsServer), responsible for spreading the new position information to the storage server (assServer), which responds by reading the audio file from the corresponding position (converted to bytes). assServer calls the client back, immediately before the new audio spurt is delivered to the audio server, to inform that the playback is starting (after the skip).

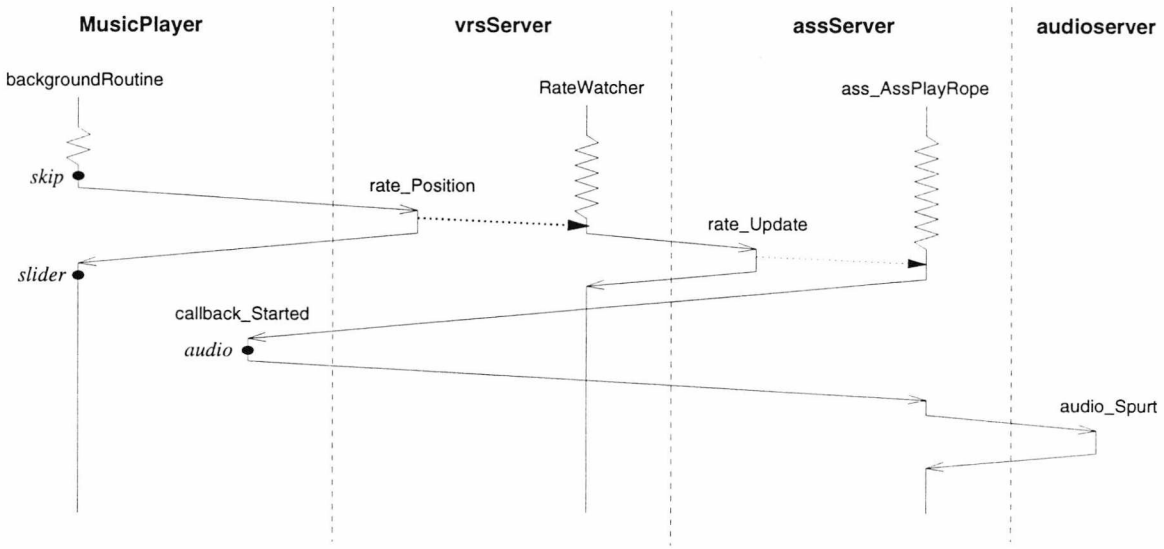


Figure 22: Achievement of the request to skip.

The other operations of interest are:

- play from position 0 (zero),
- pause at current position,
- continue from current position, and
- stop, reseting position to 0 (zero).

5.3 Measurement Modules

To make the measurements, two modules were defined: one to collect data about events during execution of the application, and another to make auxiliary measurements that will help explain the results obtained in the main experiments.

5.3.1 Run-time Module

The design of this module and the analysis of data to calculate the response times relative to the selected operations are discussed here.

Design

The module was designed to capture data, especially *time of occurrence*, about the **observable events** described in section 5.2. It was incorporated in the application, so the main concern was to have as little effect as possible on the behaviour and, consequently, on the performance of the application. The design considerations include

- the approach used to capture data about the occurrence of an event; thus, for each operation, data are captured relative to the following events: the user request for the operation, the slider positioning in the timeline of the GUI in response to the operation requested, and the corresponding response by the audio sub-system;
- the use of callbacks to indicate the audio response;
- the accuracy of the measurements, i.e. how close is the relationship between the data captured and the actual occurrence of an event; and
- the procedure to store the data captured.

The approach. In general, the approach used is to create information about an event as it happens, that is, immediately after its occurrence. The algorithm for this general approach is presented as follows.

Algorithm 1:

```
event;
get occurrence time;
record event id, occurrence time
```

However, for events that wait for certain actions to complete, time is captured prior to their triggering (see following algorithm). For events which take long to finish, it may be desirable to record the return time too.

Algorithm 2:

```
get occurrence time;
record event id, occurrence time;
event;
[get return time; record event id, return time]
```

Notice that the latter approach refers to events that synchronise with other actions, so let us call the approaches as **single-event** and **synchronisation-event**, respectively.

Callbacks. Responsibility for the audio operation, in particular, is given to the audio storage server (**assServer**). This has the ability to call back its clients to notify them of events, such as the occurrence of labels in the stream, cancellation of audio, pause, start, finish and failures. (More details of this server's role are given in section 5.5.1.) The measurements are concentrated in the clients, following the principle of not interfering much with the behaviour or performance of the system, and they rely on the callbacks to indicate that an audio event has happened.

Accuracy. The significance of the data will depend on the accuracy of the measurements. The two main points are: the timing function and the use of callbacks.

To obtain the data to represent the occurrence of events in time, the system function `gettimeofday` is called, returning the system's time expressed in seconds and microseconds. These are used to calculate the time in milliseconds, the granularity needed. The resolution of the system clock is sufficient and, with the call as near as possible to the event's instruction, the result is accurate enough to indicate events time.

Assuming that the storage server was designed to deliver callbacks as close as possible to the actual audio event, only latency needs to be considered. However, according to the RPC times measured in the auxiliary tests (see section 5.3.2 – table 7), differences are of no more than 5 ms, which does not imply any large inaccuracy (see figure 23). Additionally, system interrupts and scheduling are potential problems, but the consistency of the results obtained suggests that they are not major sources of error.

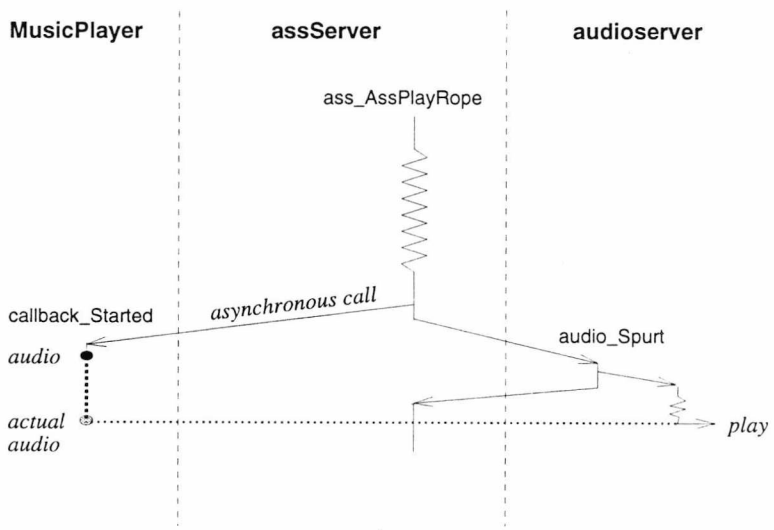


Figure 23: Accuracy of audio callbacks.

Data storage. During execution the system checks the environment variable STAT, which defines whether statistics are generated or not. The data collected in the event monitoring

are put in a buffer, which is emptied to a disk file either from time to time or when it gets full. The use of buffering follows the idea of disturbing the performance of the application as little as possible.

Data analysis

The resulting log-files can be analysed by the programs `logplot` and `logeval`. Visualisation of the events as they happened in the session is given by `logplot`, whereas operation times, averages and percentages are evaluated by `logeval`. This is responsible for separating the sequences of events corresponding to specific operation types (e.g. *skip*, *pause*, etc.), calculating their response times individually, and giving operation averages and percentages by time intervals. The results are shown in section 5.5.3.

5.3.2 Auxiliary Measurements

In addition to observing selected events while the application was running, there were measurements of times corresponding to file manipulation and process communication, as these represent the most important and the most time-consuming actions of the system.

The tests were made using a simple client-server structure, in which the client program performs the file manipulation and sends blocks of data to the server. Considerations included process location, processing power, block and message (data) sizes (granularity). The main objective is to discuss the results based on the fact that the application's audio subsystem performs the audio transfer by repeated remote procedure calls, each sending 520 bytes of data, of which 512 bytes represent audio. Also, the effect of general communication of various packet sizes in the application's performance can be examined.

The main components of file manipulation are *file opening* and *block reading*. The first

block-reading is distinguished, as it is believed to be slower than subsequent readings, and therefore, could affect the performance of the **play** operation, for example. The algorithm (in dpl notation) that represents the basic client and server behaviours is given as follows, remembering that communication is done via *remote procedure call*[1] (RPC).

Algorithm 3:

Client		**Server**
`gettimeofday(&tA,NULL);`		
/* file opening */		
`f = open(file);`		
`gettimeofday(&tB,NULL);`		
/* first read */		
`read(f,buffer,buffer_size);`		
`gettimeofday(&tC,NULL);`		
`open_time = tB - tA;`		
`read_time = tC - tB;`		
`gettimeofday(&tA,NULL);`		
/* RPC */		
`! {tB} <- server$Op(buffer)`	$\longrightarrow$	`int server_Op(in,out)`
`latency = tB - tA;`		`gettimeofday(&tB,NULL);`
`gettimeofday(&tA,NULL);`		`*out = tB;`
/* read */		`return 1;`
`read(f,buffer,buffer_size);`		
`gettimeofday(&tB,NULL);`		
/* RPC */		

[1] Clock synchronisation of different machines on the network used for testing is considered satisfactory.

```
! {tC} <- server$Op(buffer)
  read_time = tB - tA;
  latency = tC - tB;
```

File manipulation

The calibration-tests were carried out using the same audio-files employed in the application. The machine on which the files reside is referred to as "local SPARC 10". (The experiments make use of SUN's network file system (NFS) for remote access.) System load can be considered as *normal*, since the tests were done at normal working hours.

File opening. Times for a file open were much faster on the local workstation than on the other machines. Locality was more significant than processing power – compare values from the SPARC 10s (local and remote). Table 5 summarises the results.

	SPARC 2	SPARC 10	Local SPARC 10
Average (ms)	12.55	8.85	1.66
No. of experiments	35000	35000	36300

Table 5: Times for a file opening by machine type.

Block reading. Table 6 shows the average times needed to read a data-block for various grain sizes. The factors that influenced the results were machine load and processing power. The remote SPARC 10 was less loaded than the local one, and times were significantly better in the remote one in spite of the locality aspect. The fact that SPARCs 10 are faster than SPARCs 2 is confirmed by the comparison of the results obtained in these two types of machine. The results confirm that the little differences of time to read small and large blocks of data represent an advantage when using larger blocks because less disk access is required. Observe that second readings are faster than first readings due to caching in the file system.

Grain size (byte)	Reading times (ms)					
	SPARC 2		SPARC 10		Local SPARC 10	
	1st r	2nd r	1st r	2nd r	1st r	2nd r
1	16.78	0.35	7.25	0.14	7.43	0.16
4	17.02	0.39	7.30	0.14	7.58	0.16
16	17.11	0.40	7.32	0.14	7.62	0.17
256	18.02	0.40	7.51	0.15	7.68	0.17
512	18.31	0.44	7.72	0.16	7.85	0.21
520	18.33	0.44	7.71	0.17	7.71	0.21
1024	24.02	0.51	9.13	0.18	9.13	0.23

Table 6: Block-reading times by machine type.

Latency

The times needed to transfer varied-sized messages of data from one object to another are given in table 7. Once again, local transfer was the key factor. The effect of granularity

Grain size (byte)	Client remote on a SPARC 2 (time in ms)	Client remote on a SPARC 10 (time in ms)	Client local on a SPARC 10 (time in ms)
1	3.06	2.27	1.63
4	3.30	2.89	1.64
16	3.53	3.10	1.73
256	3.95	3.93	3.02
512	4.30	4.20	3.30
520	4.36	4.23	3.30
1024	4.82	4.30	3.44

Table 7: Latency times by machine type.

was observed, and the results show how expensive it is to transfer small amounts of data in comparison with packets involving 256, 512, 520 and 1024 data bytes. Also, as suggested in [Vaidyanathan 90], host speed played an important role in determining the communication and the computational performance, contributing to faster data transfer – compare the columns referring to the client on a SPARC 2 and on a SPARC 10, which is

faster than a SPARC 2.

5.4 Test Environment

The tests were performed on a local network consisting of SUN SPARCstation 2s and 10s running Unix, and communicating over a 10Mbps Ethernet, as seen in figure 24. The station where the audio files reside is a SPARC 10, labelled "local". (SPARC stations provide 64kbps PCM speech input and output.) The ANSA-trader usually runs on this local machine – although traders can be set to execute on other hosts.

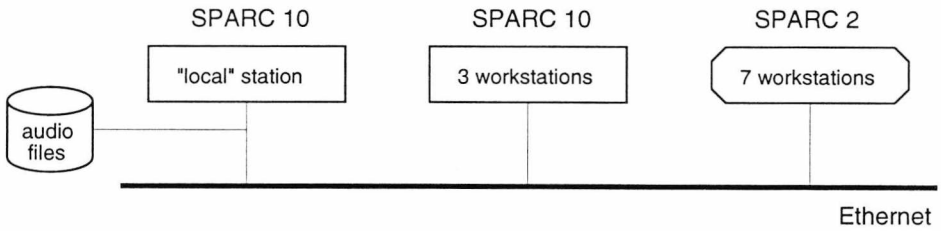


Figure 24: Test network.

5.5 Development Stages

The application development took place in three stages. Each of them achieved a different level of performance corresponding to the techniques used. The main design issues are now presented, followed by the description of the development stages.

5.5.1 Points of Attention

In the following, areas in which improvements could be made are identified. Attention is given to audio-based aspects because of the amount of data involved, which makes the

achievement of QoS requirements difficult in this area.

Audio-data manipulation

It is clear that the combination of file opening, data reading and communication represents a cost that can be reduced if appropriate policies are chosen.

Communication between the rope server and the audio storage server

The design of the audio (sub-)system has, since the early versions, been based on clear roles for each of its components. The rope server is responsible for the manipulation of the *rope* abstraction, which in practice may be composed of different audio data files, each of which is treated by the storage server, and this communicates with the audio server, the controller of the workstation's audio device, with respect to the transfer of incoming or outgoing data.

In the application, the rope server also acts as the *rate server*. The components of a rate community are all responsible for generating rate information, which must be synchronised via the server. The clients are expected to perform their rate-based tasks independently. However, due to the relationship between audio files and ropes (i.e., a single file may or may not represent a rope), the storage server sometimes needs to interact with the rope server to manipulate the audio data. Excessive communication between these two servers may degrade the performance of the application.

Audio storage server's role

During playback: in the "reading loop", besides reading data from the audio file and making RPCs to send these data to the audio server, the storage server's additional tasks include checking

(a) loss of synchronisation: difference between the rate position and the amount of data read. If an acceptable discrepancy is violated, data reading has to be adjusted – this procedure is especially useful for the skip operation;

(b) clock status: continuing, paused or stopped. In particular, when the logical clock is paused the thread's behaviour is

```
while (paused) {
      sleep(n_ms)
      keep_audio_interface_alive()[2]
}
```

(c) change of direction/speed.

In each case, appropriate actions are taken if the situation changes. These actions may impose additional delay on the communication between the storage server and the audio server.

While recording: the performance experiments do not involve audio recording, so the storage server's role in this operation is not considered here.

Why sleep? The call to `sleep` in a thread is used, basically, for two reasons:

(a) because the thread needs to wait for a certain action to happen in a different object, and cost considerations do not allow the use of a proper synchronisation mechanism like `wait(condition)`; instead, it has to use `sleep`, whose semantics is `wait(time-period)`. Moreover, `time-period` has to be guessed because there is no guarantee of when the expected action will actually happen;

[2]The audio server is prepared to discontinue service on an interface if it is not used for 60 seconds, so what `keep_audio_interface_alive` does is send an (empty) spurt to use the audio interface.

(b) in the case where actions in a sequence can be, or are required to be, separated in time (i.e. not executed immediately, one after the other) in the thread – an example of this is the procedure to keep the audio interface alive described above in (b) of the audio storage server's role during playback.

ANSAware responds to a request to `sleep(time-period)` by making the thread sleep for *approximately* the period of time required.

5.5.2 Stages

The measurement strategy was defined in parallel with the application's development, so **stage 1** represents the use of measurements when the measurement strategy first became stable and the tools were reliable. Thus some of the improvements had already been included.

Stage 1. This stage is characterised by

(a) the audio storage server interacting with the rope server for help in the following conditions:

- loss of synchronisation caused by intentional rate changes or by the data reading being too much ahead or behind the rate position,
- pause of the logical clock,
- change of direction and
- change of speed.

(b) The routine to keep the audio interface alive (see item b in **audio storage server's role**) sleeps for 25 ms within the loop that monitors the rate information;

(c) opening of the audio file was already done in the operation of rope registration, avoiding its effect of delaying the data processing;

(d) in the rate community, the storage server was the last rate client to receive broadcast information, which could increase the delay in relation to its reaction to certain changes; and

(e) before calling the operation that reads audio data, the rope server slept for 25 ms to allow the storage server to receive the clock information.

The behaviour of the operations under observation is given in figure 25 – operation calls beginning with a capital letter represent RPCs.

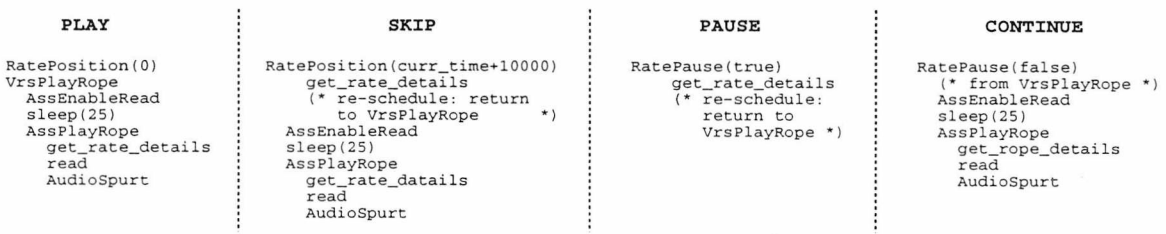


Figure 25: Behaviour in stage 1.

In figure 25, observe that **skip** and **pause** return to `VrsPlayRope`; **continue** resumes from that operation and, like **skip**, they present the same behaviour as **play** from `VrsPlayRope`. **Skip** is likely to have the worst performance amongst these user operations.

Stage 2. The features are:

(a) the problem of excessive communication between the interfaces ass (audio storage server) and vrs (voice ropes server) began to be tackled by combining

The audio storage server is prepared to react to rate changes without needing to re-schedule with the ropes server, as long as the changes imply that the same audio file will still be used – remember that a rope may be composed of more than one file. (Music ropes are, usually, formed by one single file.)

(b) The rope server no longer sleeps before calling the audio-read operation on the storage server.

Since **skip** and **continue** can take advantage of (a), only **play** can benefit from (b) – see figure 27.

```
PLAY                          SKIP                          PAUSE                      CONTINUE

RatePosition(0)               RatePosition(curr_time+10000)  RatePause(true)            RatePause(false)
VrsPlayRope                       get_rate_details               get_rate_details           (* proceed after
  AssPlayRope                     (* try to re-sync              keep_audio_itf_alive          keep_audio_itf_alive
    get_rate_details                 without returning                                       finishes *)
    read                             to VrsPlayRope      *)                                 read
    AudioSpurt                    read                                                      AudioSpurt
                                  AudioSpurt
```

Figure 27: Behaviour in stage 3.

5.5.3 Results

Stage 1

Performance of the selected user operations can be seen in figure 28; these are the average results obtained in the experiments made in this stage. As expected, **skip** presents the worst performance, while all the other operations show similar results, with **play** being slightly worse; this is not surprising, given that **skip** is the most complex of the operations to realise and **play** starts from the music player calling `VrsPlayRope` (after requesting `RatePosition(0)` – see figure 25), whereas **continue**, for example, already resumes from `VrsPlayRope`.

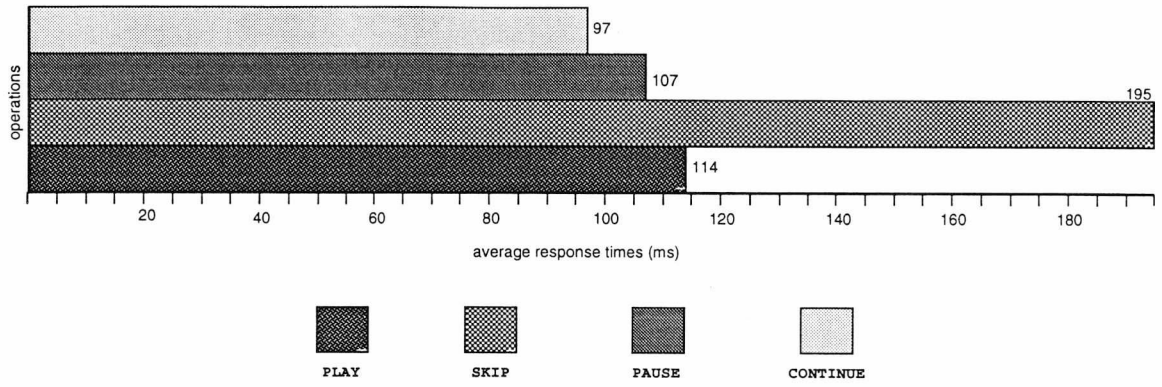


Figure 28: Performance in stage 1.

Stage 2

This stage showed a general improvement of the performance of the operations (see figure 29). The reduction of sleep-periods, both in the routine that keeps the audio interface alive and when the rope server sleeps to allow the storage server to receive the clock information, significantly contributed to the improvements. Improvements were best in relation to **play** (25%), **skip** (33%) and **continue** (56%).

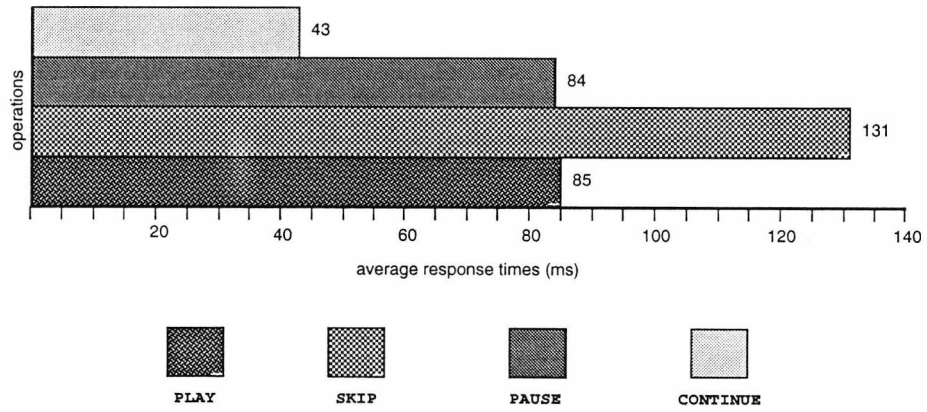


Figure 29: Performance in stage 2.

Stage 3

In this stage the target of 50-80 ms, as the maximum acceptable response times, has been achieved for all the operations being studied (see figure 30). Again, control over sleep-periods – in fact, the ropes server no longer sleeps before requesting the storage server to start reading audio data – represents an important factor in the performance of applications containing concurrent threads. In this stage, the operation **play** showed the best performance improvement.

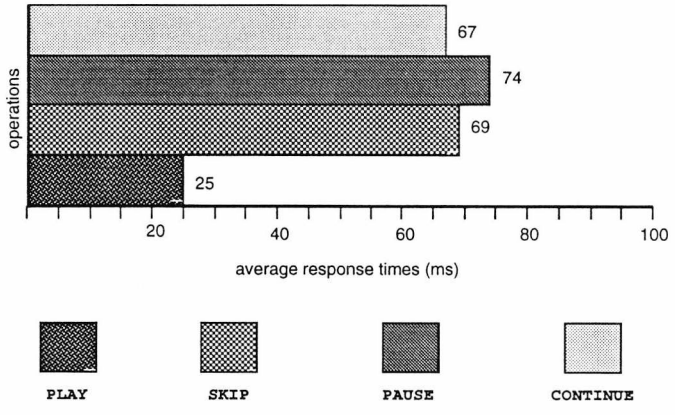


Figure 30: Performance in stage 3.

Performance improvement

Performance of distributed applications, whose components may consist of concurrent threads, can be improved through control over communication and removal of unnecessary delays. 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 should be observed. Additionally, machine loading, network traffic, etc., are factors that can result in differences in performance. (Different machines were used in the different

testing-sessions trying to minimise the influence of such factors in the results.) After considering these factors, the results shown in figure 31 clearly indicate that the improvements in the performance of the observed operations were mainly achieved through the alterations made in the application in each of the development stages. Note that the algorithm for **pause** was the least modified throughout the stages because it was the simplest and was well established since stage 1. This explains the little progress made in relation to this operation's performance, i.e. only 31% overall improvement, while **play** showed 78%, **skip**, 65%, and **continue**, 31%. Also, observe that **continue**'s performance deteriorated from stage 2 to stage 3, but this can be explained by the fact that not returning to the ropes server during a pause, the reading operation (of the storage server) tries to keep the audio interface alive, sleeping from time to time and awaking to check if **continue** has been requested, and this might make it discover the new status after some delay.

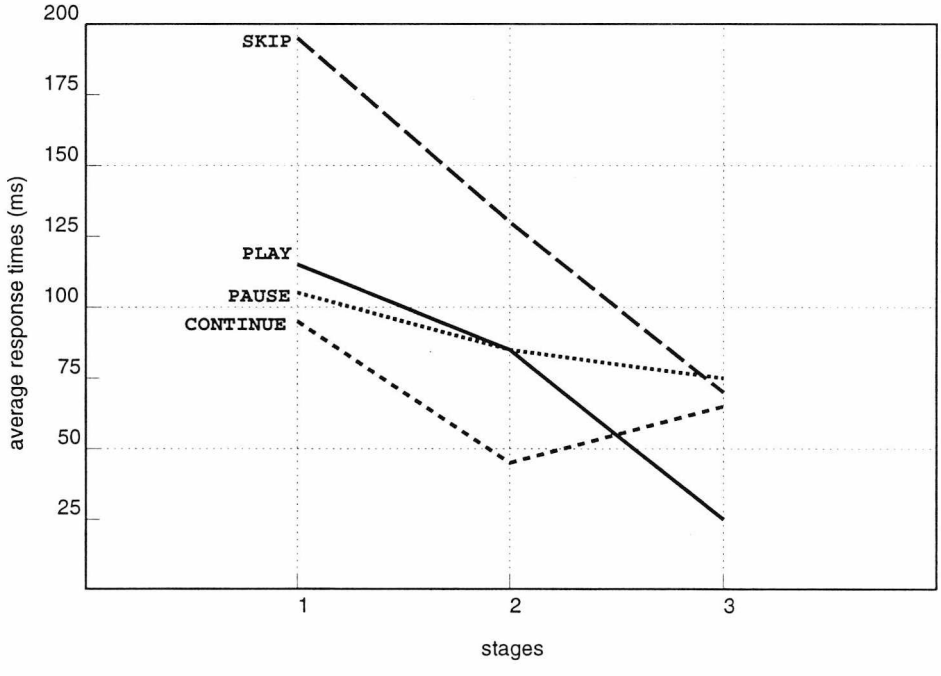


Figure 31: Evolution of performance in the different stages.

Examples

Examples of response times achieved with respect to the operations examined in the last stage of development and measurement are given in the following figures (32, 33, 34 and 35).

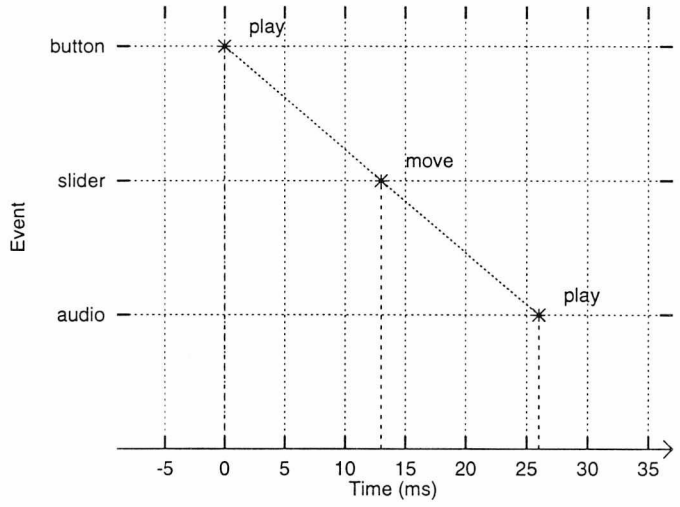


Figure 32: Example of response time for `play` in stage 3.

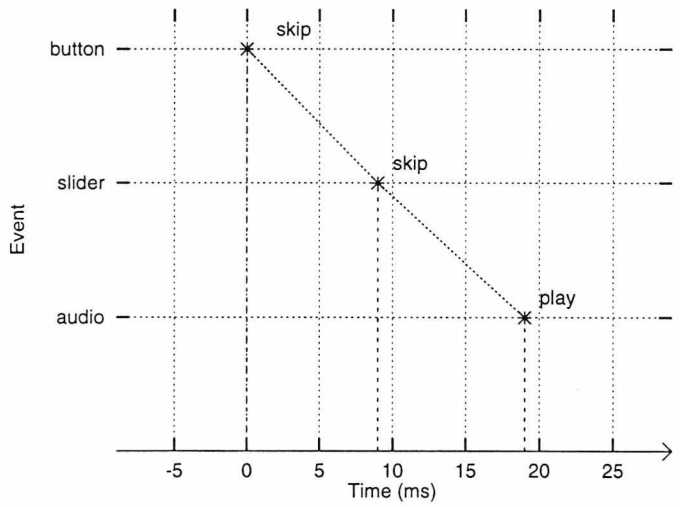


Figure 33: Example of response time for `skip` in stage 3.

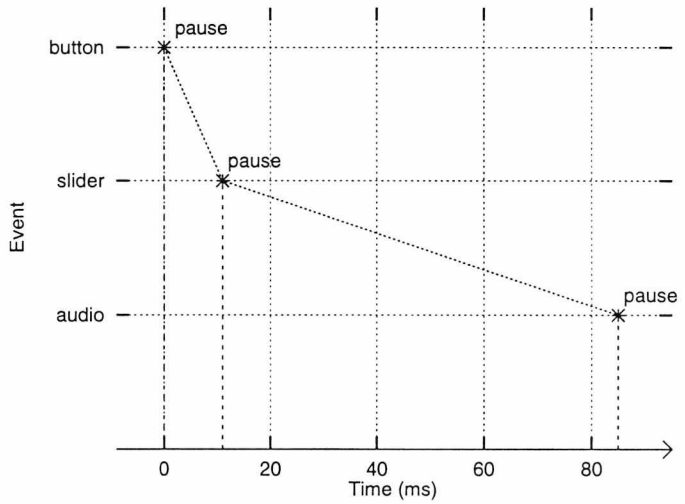


Figure 34: Example of response time for `pause` in stage 3.

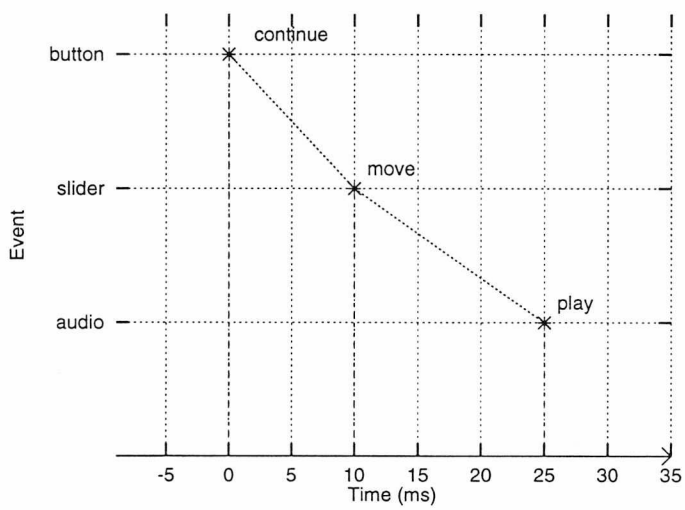


Figure 35: Example of response time for `continue` in stage 3.

5.6 Conclusions

This chapter has shown experiments made on the application implemented, in relation to response times to user requests affecting an audio presentation. A reaction of the application is a combination of actions taken by its distributed components. The mechanism developed to allow application components to interact and synchronise the logical clocks they control, forming a rate community, has been studied and has proved satisfactory in the application.

The experiments involved measurements of the times of the user request, the positioning of the slider in the timeline of the GUI, and the reaction of the audio storage server to start/stop sending data to the audio server, which controls the audio device in the workstation on which it runs. Other related measurements had to be made to study the delays for a remote procedure call (the amount of data transferred with the call across the network was also considered), and for opening a disk file and reading data blocks from it, either locally or remotely (in a LAN and supported by a network file system). Granularity in the transfer of audio data has been studied, but considered not an issue because of the existing trade-off between data transfer from an application component (storage server) to another (audio server) and buffering control within the audio server, given the low throughput of 64 kbps supported by the workstations used.

Two modules were developed to realise the measurements required: one of them involved two objects (a client and a server) to make auxiliary measurements of file manipulation and RPC times, and the other involved a strategy to measure event times in the application. Additionally, two programs were developed to calculate and analyse the response times for the operations studied. The study was realised in three stages representing modifications made in order to improve the performance of the application; some of the

alterations allowed a fine tuning of the audio sub-system.

The results presented are quite satisfactory, given a reasonably difficult target set initially. The imposed maximum delay on the order of 50-80 ms for the response times of the specified operations has been confirmed as acceptable. Although experiments relied on callbacks to represent event times, which may not be absolutely precise, the accuracy of approximate results has been discussed and suggested as reliable. The presentation of the results in stages helped show the development of the application and identify the importance of systematic alterations for the evolution of the performance.

Analysis of the results, considering the modifications that most improved response, provides the following suggestions for use in distributed applications:

- A major factor in the application's performance was the excessive message exchange between the storage and the rope servers. Better-balanced tasks may increase component independence, avoid concentration of decisions, and improve cooperation, which can be based on better-quality communication (message substance). Tools to verify and provide tasks balance in a specification would be extremely helpful to application designers.

- Applications should be given some control over system functions such as `sleep`, in addition to choosing parameters for and using these functions adequately. For example, an application should be able to interrupt a thread's sleep if a certain condition happened. In the case of the application presented, this could help improve the performance of the **continue** operation.

Chapter 6

Analysis of the Approach

This chapter analyses the approach used to model distributed continuous media applications in terms of an object-based description of support for continuous media and of software reuse. Support for continuous media is discussed in terms of rate control and stream handling, and it is compared with a different approach. The discussion about reuse involves the reuse of interface specifications, code reuse, and the reuse of the component objects presented in the model.

6.1 Introduction

This thesis has discussed the development of distributed applications involving continuous media. Therefore, it is necessary to analyse the approach used to model the applications in the terms of an object-based description of support for continuous media (i.e. synchronisation and real-time support) and of software reuse.

The approach is based on the ANSA architecture, as representative of the ODP model.

Objects offer interfaces, abstracting the services they provide, and encapsulate data and behaviour. (A distinction is made between *architectural* services, such as trading, and *application* services, provided by application objects.) Distribution can be supported by placing the objects in different locations and they can interact with each other by invoking operations defined in the interfaces provided. The recurring objects of the model proposed in this thesis are illustrated in figure 36.

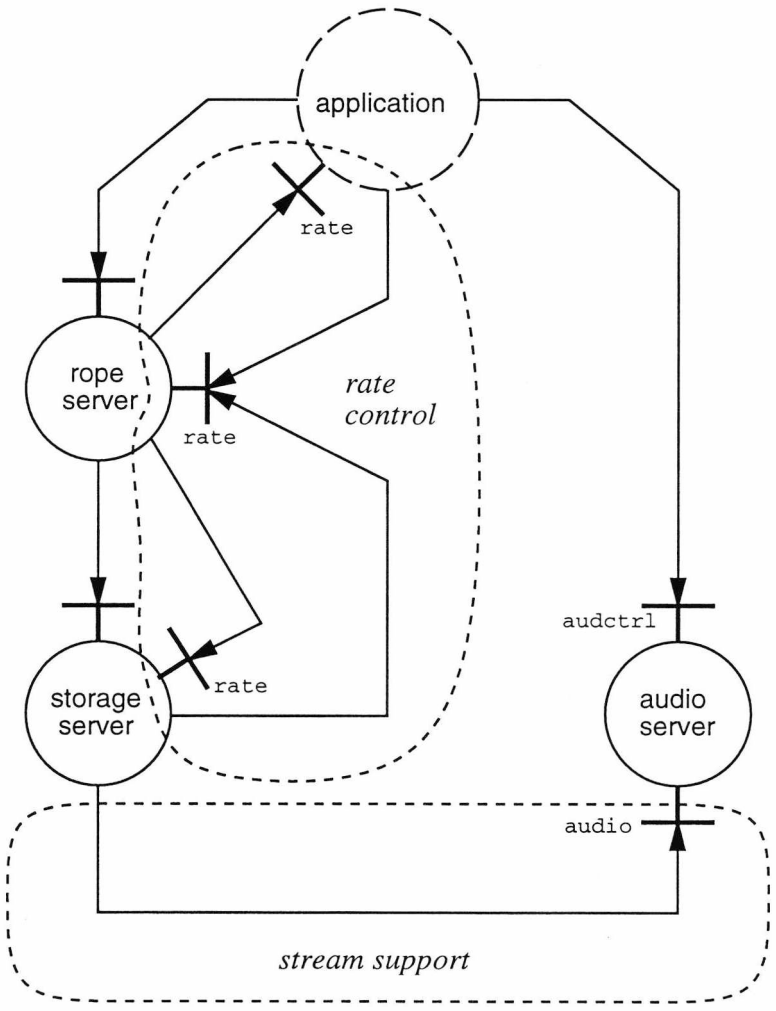


Figure 36: Structuring continuous media applications in terms of support for synchronisation and streams.

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 an audio 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.

In the model, there is thus a clear distinction between *rate control* and *stream support* – achieved by identifying a *stream transmission subsystem* involving the storage server and the device control server (e.g. the audio server), as shown in figure 36. Rate control is independent of the stream transmission subsystem and is jointly performed by the server that manages the *rope* abstraction, which represents a continuous medium sequence, 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.

6.1.1 Emphasis on Abstraction

The rope server provides an abstraction called a *rope*, based on the concept of *voice rope* introduced in [Terry 88], as a sequence of segments containing continuous media data. This simplifies the relationship between the rope server and its clients in terms of the operations

related to continuous media. The clients in an application request for a *rope* to be played, and are not concerned with the storage structures, such as a file or a sequence of frames, for example.

The benefits provided by the rope server include abstracting clients from the complexity of managing streams (a rope is a simple structure representing a continuous medium, that can be edited, stored in a database, etc.), and the possibility of attaching labels to a rope, enabling actions to be taken when the labels are found during the rope playback. From the application developer's point of view, the rope server is like a database server that provides clear operations to manipulate ropes, such as, search for a rope, build a rope, store, play, record, etc.

6.1.2 Separation between Rate Control and Continuous Media Transmission

Applications do not need to worry about continuous media transmission details, as they reuse a subsystem developed to encapsulate stream handling. Rate control is used as a synchronisation mechanism for different activities in an application. An example of an activity that needs rate control is the positioning of a slider in the time-scale of a GUI, so that it can represent time passing. Real time constraints on the continuous media need only be met at the point of presentation. Thus, for example, the exact timing of reads in the storage server is not important, so long as the correct timing is reconstructed on presentation, and there is sufficient buffering to accommodate the timing variation. What the storage server must do, however, is ensure that the mean rate and position correspond to that requested in the application. This depends on a synchronisation process which can

be carried on out-of-band with respect to the continuous medium transmission.

The components that operate the store and that manage the continuous media abstractions, plus their application clients, form a *rate community* when a particular presentation is to be controlled. For example, in an application that requires synchronisation between video and audio, the video-related and the audio-related components should participate in the same rate community, in spite of the different bandwidth requirements. In the music application described in this thesis, the annotator and the music player, together with the rope server and the audio storage server, form a rate community with respect to the music presentation, and the annotator can request another community to be formed in order to allow the presentations of music and annotation to have different rates, exploring the flexibility of the model.

These levels of abstraction and separation of concerns are what distinguish this work in terms of real-time support and reusability. In the following sections, these are each analysed in more detail aiming at demonstrating the benefits of the proposed approach.

6.2 Real-time Support

The real-time requirements in continuous media applications include

(a) *synchronisation*: which takes two forms:

   (i) *event-driven*: the system should respond to given events within a specified delay-interval – remember that we defined a maximum acceptable delay on the order of 50-80 ms for the response to user events in Chapter 5;

(ii) *multiple stream synchronisation*: here synchronisation implies the occurrence of multiple events at the same instant in time. For two or more sequences of events (e.g. sequences of audio and video frames) differences between corresponding events should not exceed a certain synchronisation tolerance;

(b) *stream handling*: a distinct form of synchronisation which enables communication and presentation at a defined rate (e.g. 30 video frames/second). Here data transmission from source to sink in a distributed environment is considered.

Our approach allocates specific functions, such as GUI control, storage, device control, etc., to application objects. Real-time support in the application can be provided by individual objects or by communities of them. Observe the decision to provide support for real time in the application level, not requiring any special service from the architecture, so that the application can be ported to a variety of similar architectures, so long as they provide general support for distributed applications.

6.2.1 Synchronisation

Multimedia applications require synchronisation mechanisms that extend the idea of controlling the order of events. Real-time synchronisation is needed to control the timings of interactions between multimedia activities. To meet such extensive requirements, some ask for the existence of synchronisation mechanisms in the communication subsystem and for operating system support [Coulson 95]. However, *open-endedness* can be quite compromised by such requirements, because the applications can become dependent on the specific systems that provide the mechanisms needed.

In contrast, our approach of including rate control in the application satisfies the requirements for supporting synchronisation and for open-endedness. The model relies on well-engineered application components, which may still require some operating-system facilities, but no special mechanism that cannot be found in any system. Moreover, separating synchronisation from communication lets the choice of rate control be separated from details of continuous media transmission and allow reuse in a number of different applications – see the discussion about reusability in section 6.3. The key to this is the concept of a *rate community*.

Rate community

A *rate community* is formed by objects interested in synchronising some of their activities. Each object defines logical clocks with respect to real time. For each set of activities to be synchronised, a rate community is established orchestrating local clocks to give global synchronisation; 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 37). Each object provides interfaces of the type rate (discussed in 4.4.1) in order to keep the clocks synchronised 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.

6.2.2 Stream Handling

Transmission of continuous media data from source to sink in a distributed environment has been considered in several works [Nicolau 90, Anderson 91, Gemmell 95, Nahrstedt 95, Coulson 95]. Some of the issues involved are definition of sample size, communication delays, transfer mechanism (RPC, etc.) and buffer management.

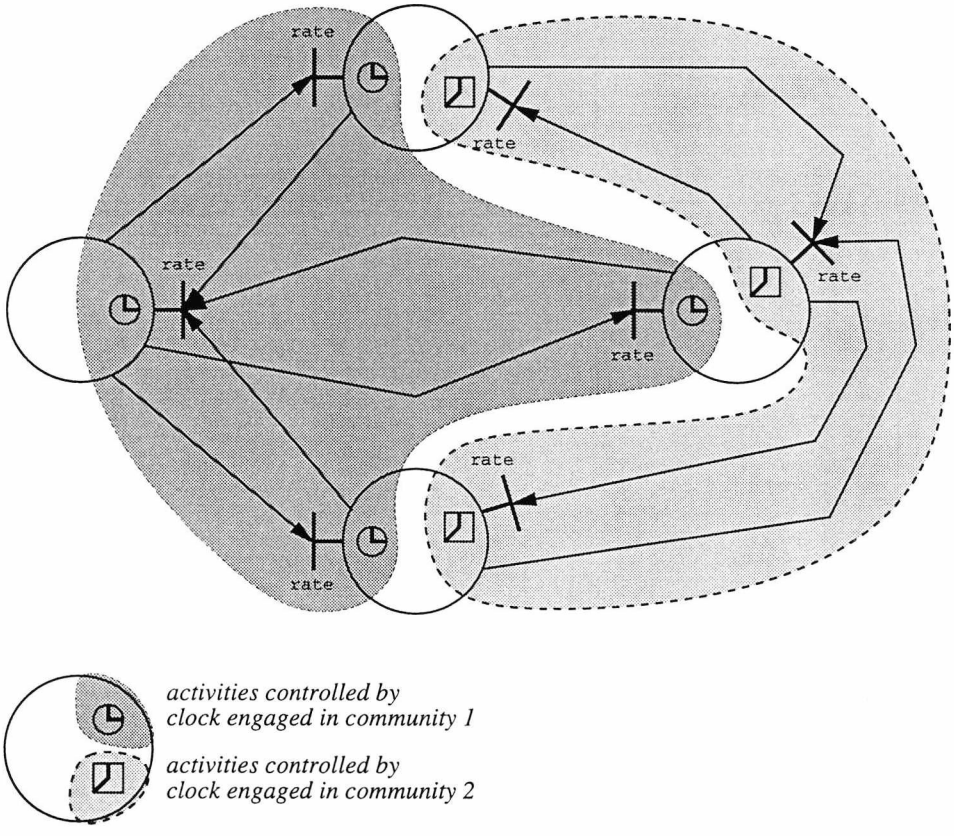


Figure 37: Rate communities in an application.

In the application discussed in this thesis, which involves the annotation of music, audio transfer occurs between the storage server and the audio server (providing the *stream support* in figure 36) and this is done by repeated RPC to provide flow control. The operation used for data transfer (**Spurt** – see 4.4.5) 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 38).

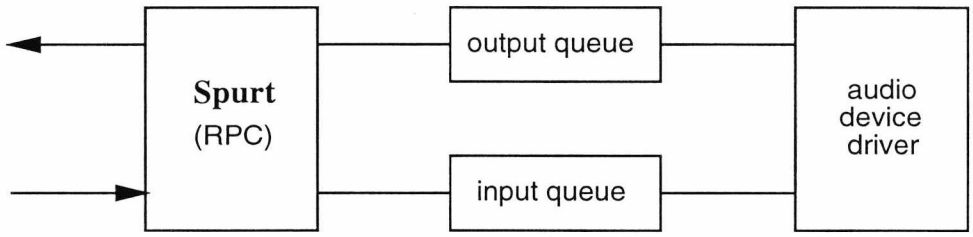


Figure 38: Buffering in the audio server.

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.

6.2.3 Summary of the Support

A summary of the real-time support, as proposed in our approach, is given below:

- real-time support is provided in the *application level*;
- rate control is implemented by a synchronisation mechanism based on *logical clocks*: a time-dependent activity in an application component is performed associated with a logical clock, which is monitored internally;
- distributed components of an application synchronise their logical clocks through a *rate community*, enabling the synchronisation of the activities associated with those clocks;
- a continuous medium sequence is represented by an abstraction called a *rope*, so that clients in an application do not get involved with details of the storage and transmission of continuous media data;
- synchronisation is separated from communication, allowing rate control to be separated from continuous media transmission – synchronisation is *out-of-band*;
- the details of continuous media transmission are encapsulated in optimised stream handling in the device control component used by applications – buffering optimisation, for example, does not need to be an issue in the development of an application.

The satisfactory performance of the test-application, as shown by the results of the measurements presented in Chapter 5, demonstrate the suitability of the support for real-time activities provided by the model.

6.3 Software Reuse

Software reuse is the process of creating software systems from existing software rather than building them from scratch [Krueger 92]. The model presented here, based on objects with clear and simple functions and clean interfaces, promotes software reuse in the development of distributed applications involving continuous media by concentrating on the reuse of distributable objects. Krueger talks about software *artifacts* that can be reused in the construction of a new software system, and classifies them as source code fragments, design structures, module-level implementation structures, specifications, documentation, etc. The techniques for software reuse involve *abstracting*, *selecting*, *specialising*, and *integrating* software artifacts.

Abstraction is the essential feature in software reuse. Concise and expressive abstractions are needed for developers to figure out what an artifact does, when it can be reused, and how to reuse it. After selecting an artifact for reuse, a developer can specialise it through some form of refinement, such as transformation, application of constraints, provision of parameters, etc. Selected and specialised artifacts are combined into a new software system through an integration framework. In the ANSA model, the existing integration framework provides facilities for interface references to be *exported* and *imported* within an application, so that the corresponding services they represent can be provided and used, once the objects that implement them are integrated into the application.

The discussion presented in the following sections analyses software reuse in the model for the development of distributed continuous-media applications in terms of

(a) reuse of interface specifications: this is the case when an object must provide a collection of functions, but the way it should behave to achieve such functionality

does not matter – only the interaction arguments and results are important;

(b) reuse of code: this refers to source code (e.g. the rate control mechanism) that can be reused in the implementation of similar activities in distinct components of an application or in similar activities in different applications; and

(c) reuse of components: this refers to the reuse of servers, or components that provide the kind of services capable of supporting similar requirements of different applications (e.g. the rope server and the audio server).

6.3.1 Reuse of Interface Specifications

Our approach requires that the interfaces in an application be clean, i.e. provide a specific, limited type of service, to support reuse, as this makes the model clear and simple. Consider the interface `callback` (page 78) as an example; in applications involving continuous media, the storage server reports events related to the playback/recording of the *rope* abstraction via an instance of this interface provided by the clients that wish to know about the events. The specification of the interface is described (in **idl** – interface description language) as follows:

```
callback:    INTERFACE =

NEEDS ass;

BEGIN

-- Operation signatures

    Failed:     OPERATION    [ vrs_rope_handle : RopeHandle;
                               reason :          AssStatus ]
                RETURNS      [ ];
```

```
--  Playback/recording cancelled by a user request
    Cancelled:  OPERATION    [ vrs_rope_handle : RopeHandle ]
                RETURNS      [ ];

--  End of rope
    Done:       OPERATION    [ vrs_rope_handle : RopeHandle ]
                RETURNS      [ ];

--  Label found in the rope
    Event:      OPERATION    [ vrs_rope_handle : RopeHandle;
                               delay :           CARDINAL;
                               action :          STRING ]
                RETURNS      [ ];

--  Playback/recording paused
    Paused:     OPERATION    [ vrs_rope_handle : RopeHandle ]
                RETURNS      [ ];

--  Playback/recording started
    Started:    OPERATION    [ vrs_rope_handle : RopeHandle ]
                RETURNS      [ ];
END.
```

The clients' reactions to the events may be different, but the interface used for callbacks has to be the one described above. Similarly, the interface `rate`[1], for the synchronisation of distributed logical clocks, is reusable in all applications which require synchronised rate control.

6.3.2 Code Reuse

This is related to design and code scavenging, as described in [Krueger 92]. In *code scavenging*, a contiguous block of source code is copied from an existing system or component.

[1]Discussed in 4.4.1 and A.3.1 (idl-description).

In *design scavenging*, a large block of code is copied, but many of the internal details are deleted while the global template of the design is retained. The effectiveness of the scavenging approach is restricted by its informality. In ideal cases, the developer is able to adapt large fragments of source code without significant modification. In his work, based on the limitations of code scavenging, Krueger mentions the second truism of software reuse:

> *For a software reuse technique to be effective, it must be easier to reuse the artifacts than it is to develop the software from scratch.*

The development of the music-annotator application presented in this thesis reused the code of the rate control mechanism very effectively, with no significant modification, because of its clarity and generality. Where rate control is applied for a number of activities, not specifically associated with continuous media, our model proposes that certain mechanisms be generalised for reuse in different applications or in components of an application.

6.3.3 Reuse of Components

Our model's notion of component reuse goes beyond the reuse of code, as discussed previously. Here the model explores the *integration framework* of ANSA, which allows active objects to connect to each other as a consequence of having references to communication interfaces. They can obtain these references by export/import via the trader or by receiving them as results of remote operations. An object that possesses the reference for an interface of another object can call operations on that interface. In this sense, an application is built as a collection of objects linked to each other via interfaces that encapsulate operations representing services that the objects can provide.

Component reuse happens in an application when it integrates objects that have been used in other applications. For this, such objects have to be designed to support a range of requirements that can be identified in different applications. For example, the rope server, providing operations on an abstraction such as a rope, representing a sequence of continuous media segments, and satisfying the requirement for maintaining a *database of ropes*, can be reused in a number of applications involving continuous media, provided that these media are represented as *ropes*. In fact, for applications involving audio, the entire audio subsystem (the rope server, the audio storage server and the audio server) can be reused by simply being *integrated* into the application, as shown in figure 36.

6.3.4 Reuse in the Model

The model, as seen in figure 36, describes the requirements for the construction of distributed continuous media applications. This model incorporates features to encourage reuse, including

- the *rope* abstraction: this allows continuous media to be kept simple, with a minimum of associated structure;
- the rope server: since it provides facilities to allow clients to invoke operations on ropes, and not on more specific and possibly more complex structures to represent continuous media, the rope server can be reused in continuous media applications that require dealing with continuous media in an abstract way;
- the storage server: a storage server is a basic requirement in applications that use stored continuous media – the storage server must be specialised for the type of continuous medium used, but its control interfaces can be general;

- the device control server: the model requires that a server (specialised by media type) exist in an application to control devices (e.g. speakers and microphone) and encapsulate buffering (as discussed in section 6.2.2);
- the *rate community* abstraction: objects in an application participate in a rate community to synchronise activities by incorporating a reusable mechanism for rate control;
- clean interfaces: this is a requirement in any model designed to support reuse.

6.4 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-synchronisation, i.e. maintaining presentation rates and synchronisation 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 thesis 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 (cf. `vrs` in appendix A.2.2) 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; a similar rope structure could, for example, be used as a basis for video 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 synchronised in a *rate community*, via a well-defined and clear interface (cf. `rate` in A.3.1). Rate-related variables, such as speed, direction (forwards/backwards) and position are controlled, allowing a logical time to be manipulated by a document author or from a user interface. The rate control mechanism can be applied to many time-varying processes, not specifically related to continuous media, such as those providing graphical feedback of progress, making it highly reusable. The response times (i.e. synchronisation delays) reported in Chapter 5 are satisfactory, demonstrating the efficiency of the mechanism;

- the continuous medium subsystem: in particular the mechanism implemented by the device control server (e.g audio server) makes the *stream transmission subsystem* (involving the storage server and the device control server – see *stream support* in figure 36) 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.

Table 8 shows a summary of the ingredients plus the advantages and disadvantages of the solutions provided.

Issue	Ingredient	Solutions Provided	Advantages	Disadvantages
Continuous media modelling	Rope server	Rope abstraction and simple interface	• Reusability • Client objects abstracted from details of continuous media manipulation • 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 synchronisation	Rate control mechanism	Logical clock, rate community, well-defined interface and clear operations	• Wide applicability • Easy manipulation	Possibly less precise than special-purpose synchronisation mechanisms
Continuous media transmission	Stream sync. subsystem	Encapsulated stream handling	• Very simple real-time links between the subsystem and the supporting sys. • Minimum application involvement • Reusability • Portability	Possibly less efficient than system-supported mechanisms, because there can be extra latency following control operations, and some additional communication cost

Table 8: Main model ingredients to be included in distributed continuous media applications.

6.4.1 Comparing with a Different Approach

The approach taken by Coulson *et al.* [Coulson 95], for example, identifies

- *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
- *programming specification* and *synchronisation in the communication subsystem plus operating system support*, in the respective viewpoints, with regard to real-time synchronisation.

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 minimised 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 synchronisation, whereas our approach has weak links, via well-known components. In our case, the separation between synchronisation and communication lets the choice of rate control be separated from continuous media transmission, making the synchronisation out-of-band; continuous medium transmission is encapsulated in optimised stream handling carried out by a stream transmission subsystem, which is integrated into the application (as shown in figure 36).

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.

6.4.2 Building Larger Systems

The fact that the separation of transport and synchronisation 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 synchronisation. 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 synchronise – 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 synchronised with the application as a whole, whereas the low level approach leads to problems of deciding which level of software owns the synchronisation hooks.

The approach proposed in this thesis is appropriate because

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

The application to the annotation of continuous media presented in this thesis served to show the efficiency of the approach, and the ingredients presented in the system proposed allow more complex applications to be designed and built.

6.5 Conclusions

This chapter has shown the suitability of the approach used to model distributed applications involving continuous media for real time support and software reuse. The approach is based on the ANSA model, which provides a language for the description of interfaces and an integration framework for the binding of application objects.

Synchronisation of time-dependent activities is achieved through a rate control mechanism separated from communication. The model does not require any special support from the communication subsystem or from the operating system – synchronisation is out-of-band. Support is provided in the application level, and thus, the approach satisfies

requirements for open-endedness. This style of design is what distinguishes our approach from other models.

Rate control is separated from continuous media transmission and can be used in a number of different applications. A time-dependent activity in an application component is performed according to the time of a logical clock set by the component, and distributed logical clocks are synchronised in a rate community formed by the components that control them.

The details of continuous medium transmission are encapsulated in optimised pieces of stream handling. Therefore, applications do not need to worry about optimising buffering, as this is mainly done in the reusable component that controls the devices for continuous medium presentation/capture.

The abstractions provided in the model, such as a rope, or a rate community, plus the fact that the interfaces are clean, making the model clear and simple, help the support for reuse. This chapter has confirmed that software reuse involves abstracting, selecting, specialising and integrating software artifacts, which can be source code fragments, design structures, specifications, etc. Our approach makes appropriate use of abstraction, selection, specialisation and integration for the reuse of interface specifications, code fragments and component objects in the development of distributed continuous media applications.

The separation of transport and synchronisation, and the possible decomposition of a rate community, allowing software subsystems to be integrated more easily, enable larger systems to be designed and built.

Chapter 7

Conclusions

This final chapter summarises the thesis. It presents the plans for future work, considering enhancements in the application, like making it support collaborative work, and porting it to another platform with a similar object model, aimed at observing the difficulties of the process of porting and the advantages and disadvantages of the various platforms. It concludes by presenting final remarks with respect to the application, networks and abstractions used.

7.1 Summary of the Thesis

This work has been a study of the development of multimedia applications using an object-oriented approach, based on the client/server model, in an open distributed environment. The basic concept domains and sources of knowledge for the work have been the object-orientation paradigm, distributed computing and open distribution platforms. Continuous media in particular, and the requirements for presentation and synchronisation of this type of media have been studied. The work has concentrated on description of components

and their interaction, and on application performance. The integration and interaction of distributed components have been facilitated by distribution platforms. A complex system, in general, is defined as a system which consists of many parts that interact with each other and which has properties resulting from such interactions. The complexity of distributed computing systems is reduced by transparencies provided by the platforms.

The thesis considers the techniques needed for the annotation of continuous media: we are all used to annotating static documents as part of our daily routine at work; with the widespread use of computers as work tools and the introduction of new types of media, documents may be in the form of audio documents, video documents, or others. The annotation of continuous media documents using voice is a subject for study, because it has, as additional ingredients, strict synchronisation requirements. The data capture, presentation and storage, database management, and GUI control can each be distributed.

The OSI Model allows the interconnection of distributed systems; a model is needed for distributed processing, preferably in an open fashion. The client/server model has been used as an approach for structuring distributed systems. Client/server applications can be created using, amongst others, the technology of distributed objects. This technology is based on the object-orientation approach, which permits objects to interact via interfaces that abstract data and behaviour; these interfaces represent the services that objects can provide. Object-oriented techniques have enhanced client/server applications, making an important contribution to a distributed processing model. Recent evidence of this trend includes initiatives such as ISO/ODP and OMG/CORBA.

ANSAware is a platform that follows the ODP Model and was designed and implemented using the object-oriented approach. It supports architectural and application objects; the most important of the architectural objects is the trader, which provides a 'rendezvous

mechanism for dynamic binding of clients to some well-known or published services'. An object has as many interfaces as needed for the services it provides. Engineering capsules are the units of execution and failure in ANSAware. Communication between objects is done via RPC, and ANSAware supports concurrency within capsules through a threads/tasks package and provides an inter-task synchronisation mechanism. The application was developed using all these basic resources provided by ANSAware to support structuring, concurrency and distribution.

The design of the application is sufficiently general to allow the use of any continuous media, including audio and video: the relationship between annotations and underlying documents is temporal, so any medium whose presentation can be controlled by observing the time (continuously) can be annotated. The mechanism that allows manipulation and control of the presentation rates, based on logical time, enables the synchronisation between the media involved. The application integrates a number of objects, some of them providing general-purpose services, which means that they can be used in different applications. In fact, the ability of ANSAware (and other platforms) to support reusability is regarded as one of its best features. Our work has successfully explored this property.

Being concerned with an application in which performance needs attention, the thesis devoted a chapter to present the results of the measurements made with respect to the synchronisation mechanism. In three stages, modifications in the application gradually improved its performance in terms of response times to user inputs. It measured how long it took for a user-action taken in a GUI, controlled by a certain object, to be responded, considering that the response depended on a combination of objects distributed over the local area network. A mechanism allows application components to synchronise their logical clocks, and the results obtained showed this to be successful. Suggestions have

been made that should be followed by any performance-critical application.

And finally, an analysis of the development approach has been made with respect to support for real-time activities, in particular the presentation of continuous media, and to software reuse in the model proposed. It concluded that the model is suitable for the construction of distributed continuous media applications.

7.2 Future Work

The following sub-sections describe directions for the continuation of this work in four topics:

- new features that should be included in the application, especially in relation to supporting collaborative work, so that some related issues can be explored;
- synchronous collaborative work, so that the synchronisation mechanism can be exploited further;
- porting the application to another platform, using a similar model, such as CORBA;
- adapting the application to handle video and high-quality audio.

7.2.1 Additional Features

The current application could be enhanced by adding:

- mechanisms to provide collaborative work,
- the possibility of sub-annotations,

- an expiry mechanism for annotations, and
- support for the "highlighting" of the parts of the underlying document which are referred to by annotations.

Collaborative work

CSCW systems allow multiple users, possibly geographically dispersed, to use computers to work together, either synchronous or asynchronously. Most of the issues related to asynchronous work can be applied in the synchronous case. In synchronous collaborative work users share the same views concurrently. Concurrency is not required in asynchronous work, but if the users work at the same time on an application, their views may not be the same [Knister 93]. To clarify the distinction between synchronous and asynchronous work, imagine the case (not necessarily computer-supported) in which two people read the same book and collaborate by providing comments on what they read; they might work in the following two possible ways:

(a) the two people observe the same page while just one of them speaks the words, so that both of them follow the same point in the narrative, and possible comments made by each one of them are listened to by both at the same time. They could be geographically separated and using the telephone, for example;

(b) the two people read different pages, possibly not knowing that they work at the same time, and record their comments, so that each one of them can have the other's comments some time later.

(a) is an example of synchronous collaborative work, while (b) exemplifies asynchronous collaboration.

The following issues can be applied, in the case of our application, to both asynchronous and synchronous collaborative work. (The synchronous collaborative annotation of continuous media is discussed in 7.2.2.)

Data sharing. In collaborative work, users should be able to access each other's annotations on the documents they share. At the same time, a mechanism should be provided to enable access control; as in a file system, a user should be able to determine *permissions*, giving access to everybody, to specific groups or to just the owner/author of the annotation. In the GUI, users should be allowed to choose between private and shared modes [Gintell 94]. The management of data sharing should be performed by the annotation server.

Consistency. With the introduction of sub-annotations [Gintell 94], the relationship between an annotation and its sub-annotations should be consistent:

- an annotation cannot be deleted[1] if there are sub-annotations referring to it;
- sub-annotations of *open* annotations should also provide shared access to the same group that access the annotation, for ethical reasons.

Because the comments of annotations cannot be altered, there is not a problem about maintaining consistency of the comments; i.e. there cannot be conflicting updates of the same comment (annotation).

Visualisation. Users should be informed of addition or deletion of annotations when they occur, i.e. the application should maintain an up-to-date view of information as it is dynamically changed by multiple users [Ben-Shaul 93]. As in [Knister 93],

[1] The delete operation should be included in future work.

any addition/deletion should be performed locally first and then broadcast, to keep response time low, and a locking mechanism should allow the GUIs to present the same views, i.e. annotation lines should appear in the same positions in the timelines of each user interface.

Identification. Users should be able to identify the authors of annotations before they run the annotations – there should be no need to recognise voices. The GUI might show the annotation icons/lines in different colours, each colour associated with a participant in the collaborating group [Karsenty 93]. A *collaborative annotation group* is associated with a specific (underlying) document; the members of the group are those who can annotate this underlying document. The colour each member chooses should be the same for at least as long as the document exists, since this determines the existence of the group.

Moreover, when the collaborative application is in single-user mode, it should behave like the single-user application [Patel 93].

Expiry mechanism for annotations

Depending on the nature of the application, annotations may be permanent or temporary. Temporary annotations should have an expiry period, which should be defined relative not to creation date but to access date. Thus, if an annotation is not accessed for a defined period of time, it should automatically be deleted or put in a different store by the application. An annotation and its sub-annotations should be considered as one for this purpose, so they should all have the same expiry period, determined by the last access to any one of them. In this case, access includes creating sub-annotations.

Highlight support

In addition to annotation timelines, there should be another kind of support to highlight the annotations and the portions of an underlying document to which they refer. This should be the case, in particular, for visual documents like a video, so that an object in the video commented on by an annotation could be highlighted during the presentation of the comment. In an early version of the application, which was an experiment with clocked sequences of images (not properly continuous), it was possible to highlight areas of the images using rectangles drawn with the cursor, as seen in figure 39.

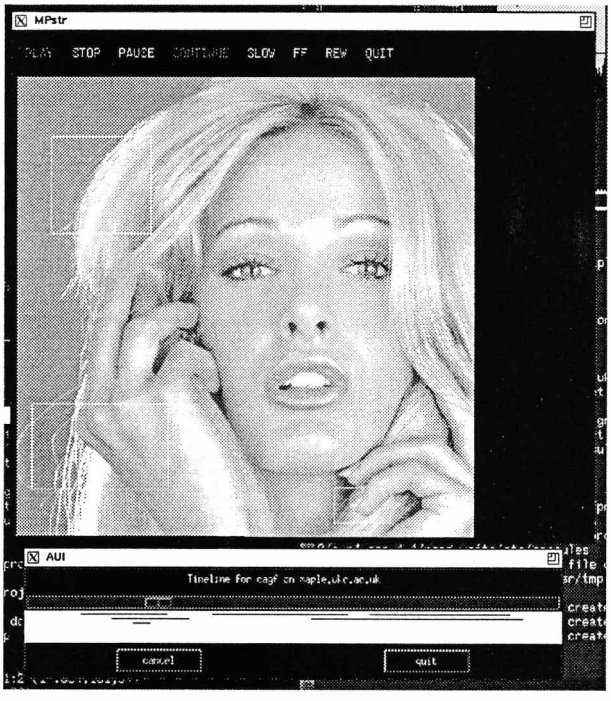


Figure 39: Annotation of clocked sequences of images showing highlights.

A tool was developed in the Network and Distributed Systems Group at UKC to allow highlighting of video [Linington 93] in a similar fashion to that shown in figure 39, i.e. using rectangles that move tracking objects in the video. This could be adapted to highlight

video on-line with annotations.

7.2.2 Synchronous Collaborative Work

The main objective for extending the application to a synchronous collaborative application is to test the synchronisation mechanisms further, since a significantly increased number of client objects would be involved. Different configurations, ranging from a centralised master that broadcasts updates to all clients, to a more cooperative community in which updates are passed from a component to its "neighbour" and so on, can be explored to check which is the most appropriate and efficient. The two basic issues are the guarantee that all participants are informed of the updates (whether the information is correct is a matter for the network protocols) and the time needed to make all participants aware of the changes.

Moreover, with respect to collaborative work performed synchronously, i.e. in which users are aware that their actions using the application are observed by others in real-time, the application has to be able to

- manage conflicts when, for example, different users decide to view different parts of the document at approximately the same time – a blocking mechanism is required;
- show results of actions to all users (also relying on the synchronisation mechanism); and
- control users joining and leaving a session, to make sure that results go to exactly those objects activated by the users actually participating in the session.

In the synchronous collaborative annotation of continuous media, synchronisation between the media involved and of user actions need to be considered in a multiuser environment (figure 40), requiring greater efforts from the synchronisation mechanism.

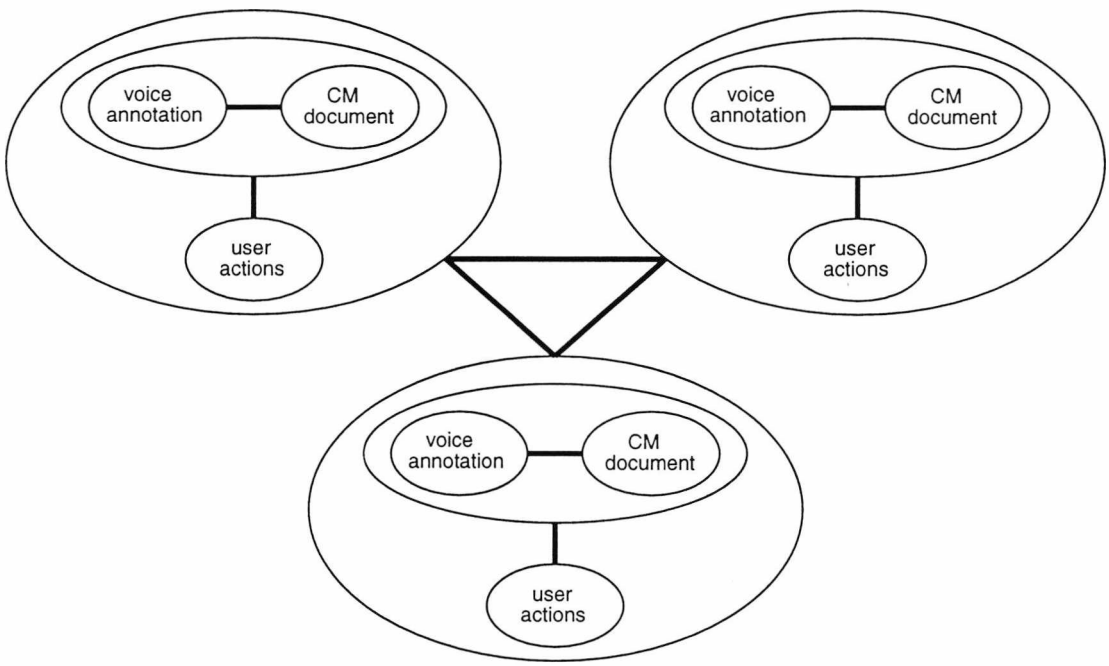


Figure 40: Synchronisation in a multiuser environment.

7.2.3 Porting the Application to Other Platform

Since the two major efforts to provide open environments for distributed processing are convergent – the groups working on ODP and CORBA collaborate on some aspects (e.g. the adoption of the IDL) and their architectures are both based on the object-oriented approach, porting the application from ANSAware to a CORBA-based platform should allow a practical comparison of the models to establish differences, similarities, advantages and disadvantages.

In principle, the adaptation should not be difficult, as the support given to applications is similar. CORBA's object services can be compared with ANSA's architectural services, although some of those provided are new, i.e. both CORBA and ANSA seem to provide similar basic services, with the former providing some additional ones. However, these might not be needed by the application. The initial comparison should be with respect

to support for the development and use of the application (e.g. component binding), and additionally with respect to the influence of the platforms on the application's performance.

7.2.4 Handling digital video and high-quality audio

The implemented application was tested over a local area network consisting of a small number of hosts under normal load. The Ethernet, with its maximum throughput of 10Mbps, has not been a serious limitation on the performance of the application, given that only uncompressed audio, with a bandwidth requirement of 64 kbps, has been used. However, faster networks such as Fast-Ethernet or ATM and compression would be required to handle video or high-quality audio.

7.3 Final Remarks

Achievements. The following are goals that have been achieved:

- **object-oriented approach**: the object-oriented approach has been used in the development of the application;
- **open distributed processing**: the application has been built on top of an open platform that supports distributed processing of communicating objects;
- **reusability**: the application has taken advantage of reusability – e.g. some components have been shown to be reusable;
- **synchronisation mechanism**: a general synchronisation mechanism has been used;

- **continuous media**: in the application, the mechanism above successfully synchronises continuous media. The implementation has demonstrated that the combination of rate control, repeated RPCs to represent continuous transfer, and buffering to avoid disruption of data presentation/capture can successfully support continuous media in distributed applications. The design of the application permits any type of continuous medium to be incorporated without much effort;

- **satisfactory response times**: the application has used resources in a way that allows satisfactory response times to be realised in the interaction between the user and the continuous media.

Additionally,

- **a measurement method** has been defined to allow response times for temporal access control operations to be measured in continuous media applications;

- **the audio sub-system has been fine tuned**, as a result of modifications made in order to improve the performance of the application, in particular, of its audio-related components, which can be reused in other applications, so that these can also benefit from the improvements; and

- **auxiliary measurements** of response times in relation to file manipulation and remote procedure calls have been made in a client/server system especially developed for this purpose.

Multiple threads and the Ethernet. Without the support for multiple threads (provided by ANSAware) it would have been very hard to develop the application, and we believe

this is true for multimedia applications in general. The application has been developed for and used in a local area network of SUN SPARCStations used for ordinary purposes and connected via the Ethernet. The SPARCStations and the Ethernet do not support the bandwidth requirements for all modes of continuous media (e.g. video and hi-fi audio), so the implementation has been restricted to telephone-quality audio (mono). In this case, the performance of the application has not been significantly compromised by the network.

And finally, the approach used and the transparencies supported by the platform have enabled a quick and easy development of distributed applications. Interfaces, the RPC paradigm and architectural services make distributed objects interact in a most natural way, like inquiring someone's telephone number through the operator, ringing and interacting with that person without necessarily knowing the person's exact location, environment and behaviour. An application can be developed using different levels of abstraction to represent its features effectively by using an object-oriented distributed model.

Appendix A

Interfaces Specification

This appendix shows the application's key interfaces according to their functions. The interfaces are specified in IDL[1] (interface description language).

A.1 Audio Input/Output

Data transfer to/from the object that controls the audio device is made through the `audio` interface.

A.1.1 The `audio` interface

```
audio : INTERFACE =

--  audio.spec - specification for the voicegram service
--
--  Copyright (c) 1990, 1991 Palantir Project
--  Modified by pfp & nl5  Aug. 1991
--
```

[1] Basic types and reserved words are entirely written in capital letters.

```
--  status values are FALSE for normal action,
--                    TRUE for shut down request
--  octet sequence is a sequence of PCM speech samples

BEGIN

    OctetSequence: TYPE = SEQUENCE OF OCTET;

--  operation signatures

    Spurt : OPERATION [
                status   : BOOLEAN;
                delay    : CARDINAL;
                samples  : OctetSequence ]
            RETURNS [
                status_out   : BOOLEAN;
                delay_out    : CARDINAL;
                samples_out  : OctetSequence ];

END.
```

A.2 Storage and Database Management

Annotation description data and audio data need to be stored and organised in databases. The `ads` interface is used to manipulate the annotation data, and the interfaces `vrs` and `ass` are used for accessing the audio (ropes) database and the audio store, respectively.

A.2.1 The `ads` interface

```
ads   : INTERFACE =

-- annotations database service interface

NEEDS AnnotType;
--    AnnotType includes the definition of Annotation
```

```
BEGIN

-- Type signature

   RequestStatus: TYPE = { FailedRequest,
                           SuccessfulRequest,
                           WrongHandle,
                           NoAnnotations };

-- Operation signatures

-- register a document with which annotations can be associated
   RegisterDoc:      OPERATION [ doc_name:    STRING ]
                     RETURNS   [ status:      RequestStatus;
                                 doc_handle:  CARDINAL ];

-- list the existing annotations associated with a document
   ListAnnotations:  OPERATION [ doc_handle:  CARDINAL ]
                     RETURNS   [ status:      RequestStatus;
                                 annot_list:  SEQUENCE OF Annotation];

   StoreAnnotation:  OPERATION [ doc_handle:  CARDINAL;
                                 annot_info:  Annotation ]
                     RETURNS   [ status:      RequestStatus;
                                 annot_name:  STRING ];

END.
```

A.2.2 The `vrs` interface

```
-- ************************************************************
-- *
-- * vrs.idl --- Specification for audio rope service
-- *
-- ************************************************************
-- *
-- *    Li Ning
-- *
-- *    Palantir Project
-- *    ----------------
```

```
-- *     Computing Laboratory
-- *     University of Kent
-- *
-- *     This version by David Barnes.
-- *     Playing a rope requires a RopeHandle, obtained by
-- *     the operation VrsRegisterRope.  A rope is played under
-- *     the control of a rate interface which
-- *     is created by the operation VrsRequestRateIf.
-- *     Different ropes may use the same rate handle, but the
-- *     length of the model only makes sense if it is related
-- *     to the length of the rope being played.
-- *
-- ************************************************************

vrs: INTERFACE =

IMPLEMENTATION IS COMPATIBLE WITH rate;

NEEDS ass;

BEGIN

-- Type signatures

    RopeSegment:     TYPE = RECORD [ ropename:        STRING,
                                     interval:       Interval ];

    RopeSegments:    TYPE = SEQUENCE OF RopeSegment;

    Label:           TYPE = RECORD [ labelname:STRING,
                                     interval : Interval ];

    Labels:          TYPE = SEQUENCE OF Label;

    LabelEvent:      TYPE = RECORD [ labelname:       STRING,
                                     action:         STRING ];

    LabelEvents:     TYPE = SEQUENCE OF LabelEvent;

    Ropes:           TYPE = SEQUENCE OF STRING;
```

```
    VrsStatus:      TYPE = { VrsOpSuccess, VrsOpFailure,
                             VrsNoMemory, RopeNameDup,
                             RSegmsToSegmsErr, VSTFull, VRHFull,
                             DumpDBErr, BadFile, UnknownRopeName,
                             LabelNameNo, AssReadErr,
                             AssEnableReadErr, AssCancelReadErr,
                             AssWriteErr, AssEnableWriteErr,
                             AssCancelWriteErr, AddLabelErr,
                             VrsNoEditLabel, VrsCantFindHandle,
                             VrsCantAddDisplay,VrsIllegalPosition,
                             VrsPositionFailure,
                             VrsNoLabelCopied,
                             -- failures to cache a rope
                             VrsReadPermissionDenied,
                             VrsWritePermissionDenied
                           };

-- Operation signatures

    VrsRequestRateIf:
         OPERATION [ ]
         RETURNS   [ status : VrsStatus;
                   -- Client rate interface
                      rate_if : ansa_InterfaceRef ];

    VrsRegisterRope:
         OPERATION [ audio_interface_ref:    ansa_InterfaceRef;
                     ropename:               STRING;
                     label_events:           LabelEvents;
                     callback_interface_ref: ansa_InterfaceRef;
                     preload:                INTEGER;
                     rate_ir:                rateRef ]
         RETURNS   [ VrsStatus; RopeHandle; CARDINAL ];

    VrsDeregisterRope:
         OPERATION [ vrs_rope_handle : RopeHandle ]
         RETURNS   [ VrsStatus ];

    VrsChangeRopeEvents :
         OPERATION [ vrs_rope_handle : RopeHandle;
                     label_events : LabelEvents ]
```

```
        RETURNS    [ VrsStatus ];

    VrsAddFilenameToHandle:
        OPERATION [ vrs_rope_handle : RopeHandle;
                    to_filename:      STRING ]
        RETURNS     [ VrsStatus ];

    VrsAddDisplayToHandle:
        OPERATION [ vrs_rope_handle : RopeHandle;
                    display_interface_ref: ansa_InterfaceRef;
                    display_scan_rate:     INTEGER ]
        RETURNS    [ VrsStatus ];

    VrsRemoveDisplayFromHandle:
        OPERATION [ vrs_rope_handle : RopeHandle ]
        RETURNS    [ VrsStatus ];

    VrsPlayRope:
        OPERATION [ vrs_rope_handle : RopeHandle ]
        RETURNS    [ VrsStatus ];

    VrsCancelPlay:
        OPERATION [ vrs_rope_handle : RopeHandle ]
        RETURNS    [ VrsStatus ];

    VrsBuildRope:
        OPERATION [ rope_segments:    RopeSegments;
                    ropename:         STRING ]
        RETURNS    [ VrsStatus ];

    -- Build a rope without the need to delete it.
    VrsBuildTempRope:
        OPERATION [ rope_segments:    RopeSegments ]
        RETURNS    [ VrsStatus; STRING ];

    VrsFileToRope:
        OPERATION [ ropename:         STRING;
                    filename:         STRING ]
        RETURNS    [ VrsStatus ];

    VrsDeleteRope:
```

```
        OPERATION [ ropename:          STRING ]
        RETURNS   [ VrsStatus ];

    VrsListRopes:
        OPERATION [ ]
        RETURNS   [ ropes:    Ropes ];

    VrsRegisterRecordRope:
        OPERATION [ audio_interface_ref:    audioRef;
                    ropename:               STRING;
                    record_interval:        Interval;
                    callback_interface_ref: ansa_InterfaceRef ]
        RETURNS   [ VrsStatus; RopeHandle ];

    VrsRecord:
        OPERATION [ vrs_rope_handle : RopeHandle ]
        RETURNS   [ VrsStatus ];

    VrsPauseRecord:
        OPERATION [ vrs_rope_handle : RopeHandle;
                    pause : BOOLEAN ]
        RETURNS   [ VrsStatus ];

    VrsCancelRecord:
        OPERATION [ vrs_rope_handle : RopeHandle ]
        RETURNS   [ VrsStatus ];

    VrsCopyLabel:
        OPERATION [ source_ropename:          STRING;
                    destinate_ropename:       STRING ]
        RETURNS   [ VrsStatus ];

    VrsEditLabel:
        OPERATION [ ropename:                 STRING;
                    label:                    Label ]
        RETURNS   [ VrsStatus ];

    VrsGetRopeDetails:
        OPERATION [ ropename:                 STRING;
                    max_number:               INTEGER ]
        RETURNS   [ VrsStatus; Labels ];
```

```
    VrsChangeDir:
         OPERATION [ directory:                STRING ]
         RETURNS   [ VrsStatus ];

    VrsShutDown:
         ANNOUNCEMENT OPERATION [ shutdown_ass : BOOLEAN ]
         RETURNS   [ ];

END.
```

A.2.3 The `ass` interface

```
-- ****************************************************************
-- *
-- * ass.idl --- Specification for audio storage service
-- *
-- ****************************************************************
-- *
-- *    Li Ning
-- *
-- *    Palantir Project
-- *    ----------------
-- *    Computing Laboratory
-- *    University of Kent
-- *
-- ****************************************************************

ass: INTERFACE =
IMPLEMENTATION IS COMPATIBLE WITH rate;

NEEDS audio;

BEGIN

-- Type signatures

    Interval:          TYPE = RECORD [ start:      LONG INTEGER,
                                       length:     LONG INTEGER
                                     ];
```

```
    Segment:            TYPE = RECORD [ filename:    STRING,
                                        interval:    Interval
                                      ];

    Segments:           TYPE = SEQUENCE OF Segment;

    LabelPosition:      TYPE = RECORD [ action: STRING,
                                        interval : Interval
                                      ];

    LabelPositions:     TYPE = SEQUENCE OF LabelPosition;

    AssStatus:          TYPE = { AssOpSuccess, AssOpFailure,
                                 AssNoMemory,
                                 AssCantFindHandle,AssOpenFailure,
                                 AssSeekFailure, AssPlayFailure,
                                 AssFilePermissionFailure,
                                 AssWriteFailure,
                                 AssCancelled,AssRopeStatusChange,
                                 AudioHangup,AudioPortDisappeared,
                                 AssReadFailure
                               };

    RopeStatus:         TYPE = { Stopped, Paused,
                                 Active, RateChange };

    RopeHandle:         TYPE = LONG CARDINAL;

    PlayDetails: TYPE = RECORD [
                                 ir_audio: audioRef,
                                 ir_display: ansa_InterfaceRef,
                                 display_scan_rate: INTEGER,
                                 ir_callback: ansa_InterfaceRef,
                                 to_filename: STRING,
                                 preload: INTEGER
                               ];

-- Operation signatures

    AssNewRateHandle:   OPERATION [ cacref : ansa_InterfaceRef ]
```

```
                    RETURNS    [ status : AssStatus;
                                 ass_rate_handle : CARDINAL ];

AssRegisterRope:    OPERATION [ PlayInfo : PlayDetails;
                                segments : Segments;
                                labels : LabelPositions;
                                vrs_rope_handle : RopeHandle;
                                ass_rate_handle : CARDINAL ];
                    RETURNS    [ AssStatus; RopeHandle ];

AssDeregisterRope: OPERATION [ ass_rope_handle : RopeHandle ]
                    RETURNS    [ AssStatus ];

AssPlayRope:        OPERATION [ ass_rope_handle : RopeHandle ]
                    RETURNS    [ AssStatus ];


AssCancelRead:      OPERATION [ ass_rope_handle : RopeHandle ]
                    RETURNS    [ AssStatus ];


AssChangeRopeEvents: OPERATION [ ass_rope_handle : RopeHandle;
                                 labels : LabelPositions ]
                    RETURNS    [ AssStatus ];

AssAddDisplayToHandle: OPERATION
                               [ ass_rope_handle : RopeHandle;
                                 display_itf_ref: ansa_InterfaceRef;
                                 display_scan_rate: INTEGER
                               ]
                    RETURNS    [ AssStatus ];

AssRemoveDisplayFromHandle: OPERATION
                               [ ass_rope_handle : RopeHandle ]
                    RETURNS    [ AssStatus ];

AssWrite:           OPERATION [ ass_rope_handle : RopeHandle;
                                interval : Interval
                              ]
                    RETURNS    [ AssStatus ];
```

```
    AssPauseWrite:      OPERATION [ ass_rope_handle : RopeHandle;
                                    pause : BOOLEAN
                                  ]
                        RETURNS   [ AssStatus ];

    AssCancelWrite:     OPERATION [ ass_rope_handle : RopeHandle ]
                        RETURNS   [ AssStatus ];

    AssEnableWrite:     OPERATION [ ass_rope_handle : RopeHandle ]
                        RETURNS   [ AssStatus ];

    AssShutDown:        ANNOUNCEMENT OPERATION [ ]
                        RETURNS   [ ];

END.
```

A.3 Rate Control

Rate control can be achieved via the `rate` interface.

A.3.1 The `rate` interface

```
rate : INTERFACE =

--
-- Palantir unified multimedia interfaces
--
-- Modified by Paul Henshaw 08.03.93
--
--      ... to include configMgmt interface.  This file now only
--      describes the synchronisation aspects of the rate interface.
--      See configMgmt.idl for details of Registration operations.
--
-- rate.spec - specification for the media rate control service
--
-- Copyright (c) 1992 Palantir Project
--
-- The reference to this interface is obtained on allocation.
```

```
-- This interface allows control of the speed and direction of
-- any clocked sequential medium, defining the logical clock
-- of the medium with respect to real time.
-- Requests are made directly or indirectly
--      by the owner of the resource
--
-- For each setting operation, the result is the value actually
-- set, which may not be the same as the request if device cannot
-- support all options
--
-- speed is the ratio (positive) of logical time to real time
-- expressed as a fraction (logical ticks/real ticks). Thus
-- (2, 3) means play at 2/3 speed
--
-- direction is true for forward, false for backward
-- note that direction can be set implicitly by negative speed
--
-- pause true is temporary pause (without change of speed/direction)
-- pause false is allows the medium to continue
--
-- position allows the medium to be spaced forward or backwards
-- the associated length is for progress displays - set negative
-- if unknown by caller
--
-- update is a convenience function combining speed, direction,
-- pause and position.
--
-- callback is an interface of type "rate"
--

-- Standard Registration operations

IMPLEMENTATION IS COMPATIBLE WITH ConfigMgmt FROM configMgmt ;

NEEDS MMtypes;

BEGIN

-- operation signatures

      ErrorNotify: OPERATION [ manager          : ansa_InterfaceRef ]
```

```
                    RETURNS [ all_ok          : BOOLEAN ];

    Speed :       OPERATION [ requestHandle   : CARDINAL;
                              requestSpeed    : Ratio ]
                    RETURNS [ responseSpeed   : Ratio ];

    Direction :   OPERATION [ requestHandle   : CARDINAL;
                              requestDirect   : BOOLEAN ]
                    RETURNS [ responseDirect  : BOOLEAN ];

    Pause :       OPERATION [ requestHandle   : CARDINAL;
                              requestPause    : BOOLEAN ]
                    RETURNS [ responsePause   : BOOLEAN ];

    Position :    OPERATION [ requestHandle   : CARDINAL;
                              requestPosn     : LONG CARDINAL;
                              requestLen      : LONG INTEGER ]
                    RETURNS [ status          : Progress;
                              responsePosn    : LONG CARDINAL;
                              responseLen     : LONG INTEGER ];

    Update :      OPERATION [ requestHandle   : CARDINAL;
                              requestPosn     : LONG CARDINAL;
                              requestLen      : LONG INTEGER;
                              requestSpeed    : Ratio;
                              requestPause    : BOOLEAN ]
                    RETURNS [ status          : Progress;
                              responsePosn    : LONG CARDINAL;
                              responseLen     : LONG INTEGER;
                              responseSpeed   : Ratio;
                              responsePause   : BOOLEAN ];

    Request :     OPERATION [ ]
                    RETURNS [ responsePosn    : LONG CARDINAL;
                              responseLen     : LONG INTEGER;
                              responseSpeed   : Ratio;
                              responsePause   : BOOLEAN ];
END.
```

A.4 Sub-systems' Interaction

The document player's interface (`docPlayer`) allows an application component (the annotator, in the case) to register its interest for interaction. The document presentation partner, therefore, presents the `pPartner` interface, which functions as a callback interface in the interaction between the application's parts.

A.4.1 The `docPlayer` interface

```
docPlayer : INTERFACE =

-- continuous media player interface

NEEDS CommonIntf;
-- includes operation NewStatus

BEGIN

   Register: OPERATION [ ir:      ansa_InterfaceRef ]
             RETURNS   [ status:  BOOLEAN;
                         handle:  CARDINAL ];

   Dereg:    OPERATION [ handle:  CARDINAL ]
             RETURNS   [ ack:     BOOLEAN ];

END.
```

A.4.2 The `pPartner` interface

```
pPartner : INTERFACE =

-- presentation partner interface

NEEDS CommonIntf;
-- includes operation NewStatus
```

```
BEGIN

   SelectedDoc: OPERATION [ doc_name: STRING ]
                RETURNS   [ ack:      BOOLEAN ];

   EndSession:  OPERATION [ ]
                RETURNS   [ ack:      BOOLEAN ];

END.
```

Bibliography

[Accetta 86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. "Mach: A New Kernel Foundation for UNIX Development". In *USENIX 1986 Summer Conference*, pages 93–113, 1986.

[Anderson 91] D. Anderson and G. Homsy. "A Continuous Media I/O Server and its Synchronization Mechanism". *IEEE Computer*, 24(10):51–57, 1991.

[APM 93a] APM. *An Overview of ANSAware 4.1.* Architecture Projects Management Ltd., Cambridge, UK, 1993.

[APM 93b] APM. *ANSAware 4.1: Application Programming in ANSAware.* Architecture Projects Management Ltd., Cambridge, UK, 1993.

[APM 93c] APM. *ANSAware 4.1: System Programming in ANSAware.* Architecture Projects Management Ltd., Cambridge, UK, 1993.

[Banerjee 87] J. Banerjee, H. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou, and H. Kim. "Data Model Issues for Object-Oriented Applications". *ACM TOIS*, 5(1):3–26, 1987.

[Ben-Shaul 93] I. Ben-Shaul, G. Kaiser, and G. Heineman. "An Architecture for Multi-User Software Development Environments". *Computing Systems (USENIX)*, 6(2):65–103, Spring 1993.

[Bertino 91] E. Bertino. "An Indexing Technique for Object-Oriented Databases". In *IEEE Intl. Conference on Data Engineering*, pages 160–170, Japan, 1991.

[Bolognesi 87] T. Bolognesi and E. Brinksma. "Introduction to the ISO Specification Language LOTOS". *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[Booch 91] G. Booch. *Object-Oriented Design with Applications.* Benjamin Cummings, 1991.

[Booch 94] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, 2nd edition, 1994.

[Burleigh 93] S. Burleigh. "ROME: Distributing C++ Object Systems". *IEEE Parallel & Distributed Technology*, 1(2):21–32, 1993.

[Campbell 93] R. Campbell, N. Islam, D. Raila, and P. Madany. "Designing and Implementing *Choices*: An Object-Oriented System in C++". *Communications of the ACM*, 36(9):117–126, September 1993.

[Cardelli 85] L. Cardelli and P. Wegner. "On Understanding Types, Data Abstraction, and Polymorphism". *ACM Computing Surveys*, 17(4):471–522, 1985.

[Chalfonte 91] B. Chalfonte, R. Fish, and R. Kraut. "Expressive Richness: A Comparison of Speech and Text as Media for Revision". In *ACM CHI'91 Conference on Human Factors in Computing Systems*, pages 21–26, 1991.

[Coats 87] R. Coats and I. Vlaeminke. *Man-Computer Interfaces: An Introduction to Software Design and Implementation.* Blackwell Scientific Publications, 1987.

[Conklin 87] J. Conklin. "Hypertext: An Introduction and Survey". *Computer*, 20(9):17–41, 1987.

[Coulson 95] 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.

[Dewan 93] P. Dewan and J. Reidl. "Toward Computer-Supported Concurrent Software Engineering". *Computer*, 26(1):17–27, 1993.

[Ferrari 92] 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.

[Fish 88] R. Fish, R. Kraut, M. Leland, and M. Cohen. "Quilt: A Collaborative Tool for Cooperative Writing". In *Proceedings of the Conference on Office Information Systems*, pages 30–37. ACM SIGOIS, 1988.

[Fox 91] E. Fox. "Advances in Interactive Digital Multimedia Systems". *IEEE Computer*, 24(10):9–21, 1991.

[Gemmell 95] D. Gemmell, H. Vin, D. Kandlur, P. Rangan, and L. Rowe. "Multimedia Storage Servers: A Tutorial". *IEEE Computer*, 28(5):40–49, May 1995.

[Ghandeharizadeh 93] S. Ghandeharizadeh and L. Ramos. "Continuous Retrieval of Multimedia Data using Parallelism". *IEEE Transactions on Knowledge and Data Engineering*, 5(4):658–669, August 1993.

[Gintell 94] J. Gintell and R. McKenny. "CSCW Infrastructure Requirements Derived from the Scrutiny Project". *ACM SIGOIS Bulletin*, 15(2):27–30, 1994. Position paper from the CSCW'94 Workshop on Distributed systems, multimedia and infrastructure support in CSCW.

[Grimshaw 93] A. Grimshaw, W. Strayer, and P. Narayan. "Dynamic, Object-Oriented Parallel Processing". *IEEE Parallel & Distributed Technology*, 1(2):33–47, 1993.

[Halasz 94] F. Halasz and M. Schwartz. "The Dexter Hypertext Reference Model". *CACM*, 37(2):30–39, 1994.

[Hardman 94] L. Hardman, D. Bulterman, and G. van Rossum. "The Amsterdam Hypermedia Model: Adding Time and Context to the Dexter Model". *CACM*, 37(2):50–62, 1994.

[Hehmann 90] D. Hehmann, M. Salmony, and H. Stüttgen. "Transport Services for Multimedia Applications on Broadband Networks". *Computer Communications*, 13(4):197–203, 1990.

[Henshaw 94] P. Henshaw. UKC ATM Video FileStore Application. Working paper, 1994. University of Kent, Computing Laboratory.

[Herbert 89] A. Herbert. The ANSA Project and Standards. In S. Mullender, editor, *Distributed Systems*, pages 391–399. Addison-Wesley, 1989.

[Herman 94] 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).

[Hindus 93] D. Hindus, C. Schmandt, and C. Horner. "Capturing, Structuring, and Representing Ubiquitous Audio". *ACM TOIS*, 11(4):376–400, 1993.

[Hoare 85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hsiao 89] C. Hsiao and S. R. Levine. "Voice Annotation in Wang Freestyle System". In *Proceedings of Speech Tech '89*, pages 365–367, 1989.

[IONA 95] IONA. The Orbix Architecture. IONA Technologies Ltd, 1995.

[ISO 81] ISO. "ISO Open Systems Interconnection - Basic Reference Model". *ACM SIGCOMM*, 11(2):15–65, 1981. Draft Proposal, DP 7498.

[ISO 84] ISO. *ISO TC97: Open Systems Interconnection - Basic Reference Model*, 1984. International Standard, IS 7498.

[ISO 92a] ISO. *Information Technology Coding for Moving Pictures and Associated Audio for Digital Storage up to about 1.5Mbit/s (MPEG).* International Standards Organization, 1992. IS 11172.

[ISO 92b] ISO. *Information Technology Digital Compression and Coding of Continuous-Tone Still Images (JPEG).* International Standards Organization, 1992. IS 10918.

[ISO 95a] ISO. *Information Processing Systems - Computer Graphics and Image Processing - Presentation Environments for Multimedia Objects (PREMO).* International Standards Organization, 1995. Committee Draft ISO/IEC 14478.

[ISO 95b] ISO. *Open Distributed Processing - Reference Model.* International Standards Organization, 1995. ISO/IEC 10746.

[Johansen 94] D. Johansen and R. van Renesse. Distributed Systems in Perspective. In F. Brazier and D. Johansen, editors, *Distributed Open Systems*, pages 175–179. IEEE Computer Society Press, 1994.

[Johnson 94] B. Johnson. A Distributed Computing Environment Framework: An OSF Perspective. In F. Brazier and D. Johansen, editors, *Distributed Open Systems*, pages 57–77. IEEE Computer Society Press, 1994.

[Karaorman 93] M. Karaorman and J. Bruno. "Introducing Concurrency to a Sequential Language". *CACM*, 36(9):103–116, 1993.

[Karsenty 93] A. Karsenty, C. Tronche, and M. Beaudouin-Lafon. "GroupDesign: Shared Editing in a Heterogeneous Environment". *Computing Systems (USENIX)*, 6(2):167–195, Spring 1993.

[Khoshafian 92] S. Khoshafian, A. Baker, R. Abnous, and K. Shepherd. *Intelligent Offices: Object-Oriented Multi-Media Information Management in Client/Server Architectures*. John Wiley & Sons, 1992.

[Knister 93] M. Knister and A. Prakash. "Issues in the Design of a Toolkit for Supporting Multiple Group Editors". *Computing Systems (USENIX)*, 6(2):135–166, Spring 1993.

[Kramer 90] J. Kramer, J. Magee, and A. Finkelstein. "A Constructive Approach to the Design of Distributed Systems". In *Proc. of 10th IEEE ICDCS*, pages 580–587, Paris, 1990.

[Kramer 93] J. Kramer, J. Magee, K. Ng, and M. Sloman. "The System Architect's Assistant for Design and Construction of Distributed Systems". In *Proc. of 4th IEEE Workshop on Future Trends of Distributed Computing Systems*, Lisbon, 1993. (7 pages).

[Kraut 92] R. Kraut, J. Galegher, R. Fish, and B. Chalfonte. "Task Requirements and Media Choice in Collaborative Writing". *Human-Computer Interaction*, 7(4):375–407, 1992.

[Krueger 92] C. Krueger. "Software Reuse". *ACM Computing Surveys*, 24(2):131–183, June 1992.

[Lea 93] R. Lea, C. Jacquemot, and E. Pillevesse. "COOL: System Support for Distributed Programming". *CACM*, 36(9):37–46, 1993.

[Li 92] N. Li. Manual pages on audio service. Working paper palantir/ukc/078, 1992. University of Kent, Computing Laboratory.

[Li 94] N. Li. "A Distributed Audio System". In W. Herzner and F. Kappe, editors, *Multimedia/Hypermedia in Open Distributed Environments*, pages 109–121. Springer-Verlag, 1994.

[Liebhold 91] M. Liebhold and E. Hoffert. "Toward an Open Environment for Digital Video". *CACM*, 34(4):103–112, 1991.

[Linington 90] P. Linington. Audio facilities for use in Palantir. Working paper palantir/ukc/050, 1990. University of Kent, Computing Laboratory.

[Linington 91] P. Linington. Revision of the audio interface. Working paper palantir/ukc/071, 1991. University of Kent, Computing Laboratory.

[Linington 92] 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.

[Linington 93] P. Linington and C. Teixeira. "Exploiting Interactive Video and Animation in Distributed Environments for the Design of Hypermedia and Graphical User Interfaces". In *VI SIBIGRAPI*, pages 213–220, Recife, Brazil, October 1993. Brazilian Computing Society.

[Linington 95] P. Linington. "RM-ODP: The Architecture". In *Intl. Conference on Open Distributed Processing*, Australia, February 1995.

[Little 90] T. Little and A. Ghafoor. "Synchronization and Storage Models for Multimedia Objects". *IEEE Journal on Selected Areas in Communications*, 8(3):413–427, 1990.

[Little 94] T. Little. Time-based Media Representation and Delivery. In *Multimedia Systems*, chapter 7, pages 175–200. Addison Wesley, 1994.

[Lorin 90] H. Lorin. "Application Development, Software Engineering and Distributed Processing". *Computer Communications*, 13(1):4–16, 1990.

[Mackay 89] W. Mackay. "EVA: An Experimental Video Annotator for Symbolic Analysis of Video Data". *ACM SIGCHI Bulletin*, 21(2):68–71, 1989.

[Magee 89] J. Magee, J. Kramer, and M. Sloman. "Constructing Distributed Systems in Conic". *IEEE Transactions on Software Engineering*, SE-15(6):663–675, June 1989.

[Meyer 93] B. Meyer. "Systematic Concurrent Object-Oriented Programming". *CACM*, 36(9):56–80, 1993.

[Meyer-Boudnik 95] T. Meyer-Boudnik and W. Effelsberg. "MHEG Explained". *IEEE Multimedia*, 2(1):26–38, Spring 1995.

[Mullender 89] S. Mullender. Interprocess Communication. In S. Mullender, editor, *Distributed Systems*, pages 37–64. Addison-Wesley, 1989.

[Mullender 93a] S. Mullender, editor. *Distributed Systems.* Addison-Wesley, 2nd edition, 1993.

[Mullender 93b] S. Mullender. Kernel Support for Distributed Systems. In S. Mullender, editor, *Distributed Systems*, pages 385–409. Addison-Wesley, 1993.

[Myers 85] B. Myers. "The Importance of Percent-done Progress Indicators for Human-Computer Interfaces". In *Human Factors in Computer Systems – CHI'85 Conference Proceedings*, pages 11–17, ACM. New York, 1985.

[Nahrstedt 95] K. Nahrstedt and R. Steinmetz. "Resource Management in Networked Multimedia Systems". *IEEE Computer*, 28(5):52–63, May 1995.

[Nicolau 90] C. Nicolau. "An Architecture for Real-Time Multimedia Communication Systems". *IEEE Journal on Selected Areas in Communications*, 8(3):391–400, April 1990.

[Noll 91] J. Noll and W. Scacchi. "Integrating Diverse Information Repositories: A Distributed Hypertext Approach". *Computer*, 24(12):38–45, 1991.

[OMG 92] OMG. *The Common Object Request Broker: Architecture and Specification.* Object Management Group, 1992. OMG Document Number 91.12.1, Revision 1.1.

[Orfali 95a] R. Orfali and D. Harkey. "Client/Server with Distributed Objects". *BYTE*, 20(4):151–162, 1995.

[Orfali 95b] R. Orfali, D. Harkey, and J. Edwards. "Intergalactic Client/Server Computing". *BYTE*, 20(4):108–122, 1995.

[Panzieri 93] F. Panzieri and R. Davoli. Real Time Systems: A Tutorial. Technical Report UBLCS-93-22, University of Bologna LCS, Italy, 1993.

[Patel 93] D. Patel and S. Kalter. "A UNIX Toolkit for Distributed Synchronous Collaborative Applications". *Computing Systems (USENIX)*, 6(2):105–133, Spring 1993.

[Protogeros 90] A. Protogeros and E. Ball. "Traffic Analyser and Generator - Part 1: High-speed traffic capture for IEEE 802.3/Ethernet networks". *Computer Communications*, 13(7):407–413, 1990.

[Rozier 88] 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.

[Schneiderman 83] B. Schneiderman. "Direct Manipulation: A Step Beyond Programming Languages". *Computer*, 16(8):57–69, August 1983.

[Schroeder 93] M. Schroeder. A State-of-the-Art Distributed System: Computing with BOB. In S. Mullender, editor, *Distributed Systems*, pages 1–16. Addison-Wesley, 1993.

[Shepherd 90] D. Shepherd and M. Salmony. "Extending OSI to Support Synchronization required by Multimedia Applications". *Computer Communications*, 13(7):399–406, September 1990.

[Shneiderman 92] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley, 2nd edition, 1992.

[Singhal 91] M. Singhal and T. Casavant. "Distributed Computing Systems". *Computer*, 23(8):12–15, 1991.

[Snyder 93] A. Snyder. "The Essence of Objects: Concepts and Terms". *IEEE Software*, 10(1):31–42, 1993.

[Stefani 92] J. Stefani, L. Hazard, and F. Horn. "Computational Model for Distributed Multimedia Applications based on a Synchronous Programming Language". *Computer Communications*, 15(2):114–128, 1992.

[Steinmetz 90] R. Steinmetz. "Synchronization Properties in Multimedia Systems". *IEEE Journal on Selected Areas in Communications*, pages 401–412, 1990.

[Steinmetz 96] R. Steinmetz. "Human Perception of Jitter and Media Synchronisation". To appear in IEEE Journal on Selected Areas in Communication, 14(2), February 1996.

[Stenzel 94] 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.

[Tanenbaum 90] A. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, A. Jansen, and G. van Rossum. "Experiences with the Amoeba Distributed Operating System". *CACM*, 33(12):46–63, 1990.

[Tanenbaum 92] A. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 1992.

[Terry 88] D. Terry and D. Swinehart. "Managing Stored Voice in the Etherphone System". *ACM Transactions on Computer Systems*, 6(1):3–27, 1988.

[Thomas 85] R. Thomas, H. Fordsdick, T. Crowley, R. Schaaf, R. Tomlinson, V. Travers, and G. Robertson. "Diamond: A Multimedia Message System built upon a Distributed Architecture". *Computer*, 18(12):65–77, 1985.

[Tilley 91] S. Tilley and H. Muller. "INFO: A Simple Documentation Annotation Facility". In *ACM 9th Intl. Conference on Systems Documentation*, pages 30–36, 1991.

[Trigg 88] R. Trigg. "Guided Tours and Tabletops: Tools for Communicating in a Hypertext Environment". In *ACM CSCW'88*, pages 216–226, 1988.

[Vaidyanathan 90] P. Vaidyanathan and S. Midkiff. "Performance Evaluation of Communication Protocols for Distributed Processing". *Computer Communications*, 13(5):275–282, 1990.

[van Nes 92] F. van Nes. Design and Evaluation of Applications with Speech Interfaces – Experimental Results and Practical Guidelines. In *Methods*

and Tools in User-Centred Design for Information Technology, pages 281–297. Elsevier, 1992.

[Vin 91] H. Vin, P. Zellweger, D. Swinehart, and P. Rangan. "Multimedia Conferencing in the Etherphone Environment". *Computer*, 24(10):69–79, 1991.

[Vissers 91] C. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification Styles in Distributed Systems Design and Verification. In *Theoretical Computer Science 89*, pages 179–206. Elsevier, 1991.

[Wegner 87] P. Wegner. "Dimensions of Object-Based Language Design". *ACM SIGPLAN Notices (Special Issue)*, 22(12):168–182, 1987. Also in Proceedings of OOPSLA'87.

[Wegner 90] P. Wegner. "Concepts and Paradigms of Object-Oriented Programming". *OOPS Messenger*, 1(1):7–87, 1990.

[Yamazaki 93] S. Yamazaki, K. Kajihara, M. Ito, and R. Yasuhara. "Object-Oriented Design of Telecommunication Software". *IEEE Software*, 10(1):81–87, 1993.

[Zellweger 92] P. Zellweger. Toward a Model for Active Multimedia Documents. In M. Blattner and R. Dannenberg, editors, *Multimedia Interface Design*, pages 39–52. Addison-Wesley, 1992.

[Zhang 91] H. Zhang and S. Keshav. "Comparison of Rate-Based Service Disciplines". In *ACM SIGCOMM'91*, pages 113–121, September 1991.