Sharing Multimedia Information: a basis for assisted remote training

Luis Bernardo and Paulo Pinto
Inesc/IST, R. Alves Redol, 9 P-1000 Lisboa Portugal
{lflb,pfp}@inesc.pt

Abstract: *Multimedia cooperative work has several requirements not supported by existing shared systems. A multimedia distributed platform, DIMPLE, suitable for constructing interactive and distributed multimedia applications, was enhanced to support shared multimedia applications. This paper describes these enhancements and presents a typical application of shared synchronized access to an information retrieval system. The open nature of DIMPLE and its constructive type schema were important to incorporate an off-the-shelf videoconferencing system with minimal changes.*

1. Introduction

Most of the multimedia applications reported recently are concerned with videoconferencing systems; define frameworks for cooperative work (CSCW); or tackle the problem of information retrieval in its various aspects of synchronization and presentation.

One important issue was always neglected: the support for more than two users in a very versatile way. More specifically, it would be good if users could join "working sessions" and leave as they please. Once in the session they would participate synchronously with everybody else that was present, and applications would not be aware of such.

A working session can be anything from videoconferencing to sharing any kind of application even (and specially) if it is a multimedia information retrieval system. In this case, users can see the same information at the same time and browse as if only one user was in the system. This is particularly interesting for applications in which a tutor supervises some learning process. For instance, the replacement of a spare on an industrial machine. The maintenance person would learn from a multimedia document. If (s)he has some doubts (s)he could start a shared session with the support responsible person of the manufacturer. The support person would follow the session and would explain further details using the videoconferencing, or would influence some browsing decisions. The support person would join exactly at the place where the problem was and could leave anytime after that (joining again, etc.).

For non-multimedia data a well-known technology to share applications is SaredX. However, as it works at rendering level, it is prohibitive for multimedia applications due to the amount of data. Other aspects of SharedX are also a handicap, as for instance the use of TCP for continuous real-time signals.

Furthermore, if the basic concept for a user is not suitable, the action of joining another user to the sharing system can create problems at communication level and also at control level inside the application. It creates unnecessary complexity.

This paper describes a shared multimedia application built on a distributed multimedia platform, DIMPLE. The DIMPLE platform is an extended version of the architecture described in [8] and in [9]. The multimedia support is given by the addition of multimedia sockets and plugs to the ANSAware system [2]. The concept of a multimedia object is unique, regardless of the number of end points it has internally, giving a unique control interface to the application. Users can just plug into the existing objects of an application sharing all the current environment of the application.

2. The DIMPLE Platform: Multimedia and Sharing Extensions

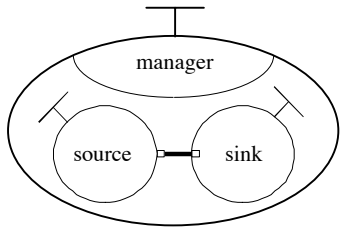

Figure 1 - MMobject architecture

2.1. Object Model

Object-based technology was used to define multimedia entities is and the model is an extension of the one presented in [9]. A multimedia object, a MMobject, is formed by a set of *component* objects (sources, sinks and filters) and by a manager which offers a unique interface to the entities using it (see figure 1).

There is a neat distinction between control of the object and the (multimedia) data the object is able to transport. Internally, the manager reflects this distinction by being subdivided in the *synchronization manager* and the *topology manager*. They are responsible for the control of the MMobject, and for the multimedia data topology, respectively.

2.1.1. MMobject control

The control of the MMobject is performed via its unique interface, and (possibly) mapped internally onto the control interfaces of the components. The control interactions are made of events and actions between objects. Actions are submission of invocations. Events support the returning of values and the notifications of changes to the internal state of the object. Control operations are not so time critical as multimedia data transfer, and have different interaction requirements. An RPC-like mechanism is used (the ANSAware plugs and sockets).

The platform was conceived to deal with a wide range of entities, with different control interfaces. To avoid the use of operation overloading, that results unnatural to the programmers, the actions, related events and internal states were grouped in *statuses*. All MMobjects (and component objects) have a basic common set of statuses (*Life* and *Context* statuses), and further statuses which are dependent of their types (audio objects have *volume* status, video objects have *brightness* status, etc.). The basic group supports the instanciation and the basic management of the objects. The interface of the *Life status* has an action to instantiate the object, *Prepare*, and another one to abort the instance, *Dismiss*. The interface of the *Context status* supports the play and stop of the object; its destruction; the registration of interest in events; the provision of references to the other status of the object; and a general event dispatcher. It provides a unique identifier to the MMobjects in the platform. Components also have the *Context status* and have a similar instantiation method, reunited in *LifeComp* (the tasks performed are slightly different from the ones for MMobjects).

The addition of more status to existing objects is allowed because all the type information is checked at run-time. When a new set is added, a new MMobject type is, in fact, created, and it should be conformant with the old one for the common statuses. Applications should work even if a status they need is not supported (this is consistent because new statuses add no synchronizing features and simply side control features, such as volume, etc.). It is obvious that old applications do not make any use of new status. With this mechanism, evolution over time is guaranteed.

The synchronization manager is responsible for managing the status whose functionality is distributed by several components. However, if a status is only supported by one component (e.g. the *speed* status in a video source) then the status management will be delegated to the component. For the controller of the application it is irrelevant, because he receives an interface reference that is called transparently, without knowing what is the real server he is using.

2.1.2. Multimedia data

Multimedia data is an internal matter of the MMobjects and is transported using connections between sources and sinks (possibly with filters in between for data conversion). The *topology manager* manages all the necessary issues, acting as a Binding Object (as defined on the computational model of the RM-ODP[6]). It offers one controlling interface to the exterior, *topology status*, and uses internally some control interfaces of the MMobject components, *port status*. The *port* described now but for the *topology* it is necessary to introduce some concepts, and it will be described later on. All the controlling operations are RPC-like for the reasons explained above.

Sinks and sources are connected to transport multimedia data using high-performance channels (HPC), represented in figure 1 by small rectangles in the components. *Port status* is used to control these HPCs. HPC are typed entities and polymorphism was used to define different controlling. They all descend from *port*, which has the operations (unchanged by subtyping):

Connect (...)
Disconnect (...)

These operations allow the connection and disconnection of HPC between component pairs. They can also be used for multipoint connections, formed by a set of 1:1 relations with a common HPC control.

The sub-type categorization of the channels reflects two levels: transport and presentation. The first divides the space according to the transport algorithms used. As a general rule subtypes add a *connectProtocol* operation to the *port status* to establish the HPC. Some examples are:

tcpport	(uses TCP/IP directly)
udpport	(uses UDP/IP directly)
xtpport	(uses XTP directly)
ansaport	(uses ANSAware channels as HPCs)

The second level of type definition is used to characterize both the kind of information which is transported over the connection and the real-time enforcing algorithm as well. Encoding formats can be G.711 audio, mpeg video, jpeg pictures, etc.; Enforcing algorithms include the way data is exchanged (if sources are pushing data, or sinks pulling it), the format used to transport data (MPEG interchange format, or private), etc. The current system has an injective relation from this second level to the first (MPEG private video only uses UDP) but it need not be like this. Examples of this second level are:

*G.*711	(uses a private frame format with G.711 audio)
*mpeg_*OSI	(uses the MPEG interchange format)
mpeg_private1	(uses a private frame format for MPEG with a real-time protocol)

Type checking is performed at run-time, with the help of a Type Repository. Compatibility rules are equality of types, or other *isClassOfPortTypes* relations stored at the Type Repository. The system checks if compatibility exists before connecting *ports* (and HPCs after them).

This type schema is very powerful and open because, as it will be seen on the implementation example, it can integrate alien systems very easily. The example uses the transport facilities of a video conferencing system (called Berkom-MMC [1]) for the audio, i.e., Berkom-MMC acts as an HPC channel. The integration consisted of the definition of another *port*, called *mmcport*, and its *Connect* and *Disconnect* operations use the MMC audio transport as its HPC.

2.2. Sharing Objects

The extension to a shared environment is straightforward. As HPCs are MMobjects´ internal matters, the only thing it is needed is to allow several connections between a source and various sinks, for instance (figure 2). *Connect* and *Disconnect* are present on every port of the component objects, and they support the dynamic control of connections between ports.

This new view of MMobjects introduces some requirements to the object model, at both the MMobject synchronization control and the multimedia data control levels.

2.2.1. MMobject synchronization control

In terms of general synchronization control, nothing is changed: control is exercised via the set of status supported by the MMobject (and its components) and the shared object is essentially the same with a unique interface to the exterior. The new component control statuses that are created are dealt with by the MMobject manager, internally to the object. Specific control, however, is not so simple. Certain aspects used to be handled by just one component (delegation) in the unshared version and now a cooperation between components must exist. This is still hidden to the exterior because the controller of the MMobject used to have a reference for an interface and continues to have one now (but probably the server is the MMobject itself).

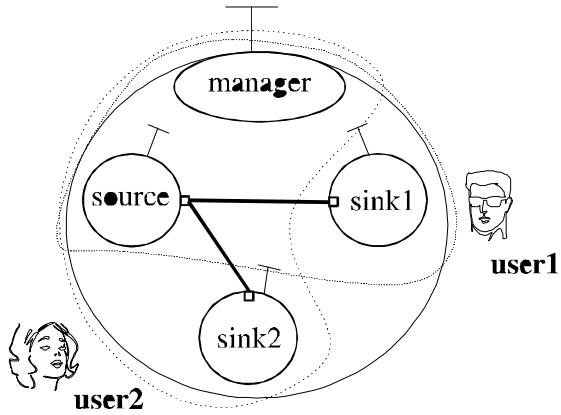

Figure 2 - MMobject shared by two users

A different issue happens when the application needs to control the various users in the system independently. There are several examples: the presentation of a document with the language of the text and the audio objects selected individually per user; a floor control algorithm in a videoconferencing system; the semantics of a user token request button; etc. The approach taken was to avoid having component visibility from the exterior as much as possible, making the scale up of the system dependent on the ability of the MMobjects to control new users. All the situations implemented can fall into three categories:

- the relevant commands can be described using some kind of language (for instance relating properties of the components to link with object properties selecting the language). Users are distinguished by properties.
- the relevant commands can be expressed in high-level policies selected through the unique control interface (the floor holder gets its audio with a higher volume). There is no awareness of users.
- Some specific control status (operations and events) are parameterized by the user id (for instance the machine name) when this information is needed for the specific algorithms.

Internally to the MMobject, "user connections" (represented in figure 2 by the dotted lines) would be seen by the MMobject manager as an array of destinations (users) and a related array of interfaces. Different distributed algorithms use these arrays to perform their tasks. Whether or not all the possible situations can be handled in this way, and do not require the components to be visible to the exterior of the MMobject, is difficult to say.

2.2.2. MMobject topology control

If the control of the MMobject topology was based on individual components, it would have been still possible to manage a set of possible destinations (user sites) but would have been a complex task. There would have been the need to control every individual component instantiation and connection. Consequently, the knowledge of the current destinations would have had to be external to the MMobject.

To simplify this task, it was introduced a level of abstraction to describe the degree of replication of each component. Taking the example of figure 2, the MMobject is built by a new sink and the same

source for each new user. The source is shared by all users (so it has only one replica independently of the users), and the sink has one replica for each user. So, the topology control can be done in user terms and derive the corresponding connections based on the replication level of each component. It is a similar approach to the subsection above making users visible and not components.

Various replicas of the same component type can be seen as belonging to a **component array**. This array simplifies the handling of the topology operations. For instance, when a component has a replica for each destination, then it is just as if it has a "single" connection to the array, in terms of MMobject topology control. Building and destroying individual components is just aggregating or separating the elements with regard to the "single" connections. The *topology status* offers a set of operations to control the topology including adding and removing users (actions *add_destination* and *drop_destination*).The MMobject reacts according to the component array type and all components that need different replicas for each user have new components instances created.

The degree of replication of a component per destination (user) can be classified in three types, reflected also on the component array types: if the component only has one replica, independently of the number of active destinations, it is called "*single*" (for example a video database source); if the component has a single replica for each destination it is a "*replicated*" component (for example a local video window in a video distribution system. See figure 3a); finally, if the component has N replicas per destination it is a "bi-replicated" component (for example a local video window in a video conference system). The bi-replication is needed because if a replicated source that broadcasts to every destination must be connected, then a sink per destination for that particular source is need (see figure 3b).

These component arrays support the definition of an arbitrary complex dependency between the number of destinations and the internal structure of the MMobject. Figures 3a and 3b show two of the most representative cases: Various users accessing a common document (each sink is a replica) and a videoconferencing with sources for each user (audio, video, etc.) and their sinks for each user (including the local one).

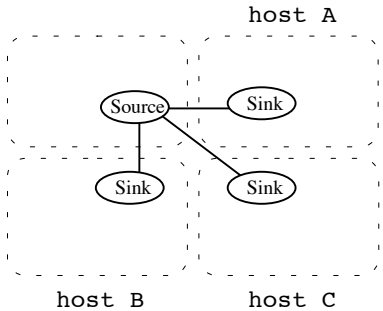

Figure 3a - video multicast MMobject

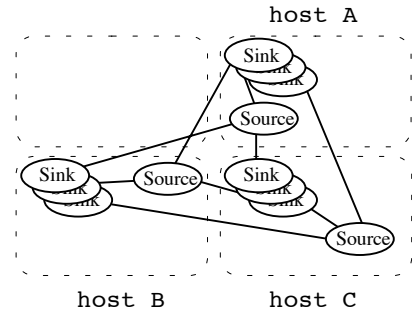

Figure 3b - video MMobject in a videoconference

The use of component arrays solves almost all the problems encountered. Special hardware devices, however, have specific features which do not fit in such a regular handling of the problem. For instance, the XVideo board from Parallax insists on having a video window present to be able to grab the frame. For this case, there is no need for a local sink. If a suitable notation to describe this topology constraint is not found, a lower level interface would have to be provided giving component visibility to the exterior of the MMobject (This issue is still under research).

Internally to the MMobject, the *add_destination* and *drop_destination* actions of the *topology status* are mapped into: (a) calls to the *Prepare* (*LifeComp status*) operations on component template servers to create instances; and (b) subsequent calls to the *Connect* operations of each one to create the binding between them (performed internally by the *connectProtocol* to create the HPCs). The *topology manager* gets to know the *port status* references of the components from the return parameters of the *Prepare* operation. It could also obtain it via the *Inform* operation of the *Context* status, as with any other status.

2.4. Using the control features with a specification language

The exchange of control action invocation and flow of events between MMobjects (i.e., the application itself) can be efficiently described using a process algebra based language. Figure 4 shows the structure of language used to define the applications, and the structure of the MMobject declaration.

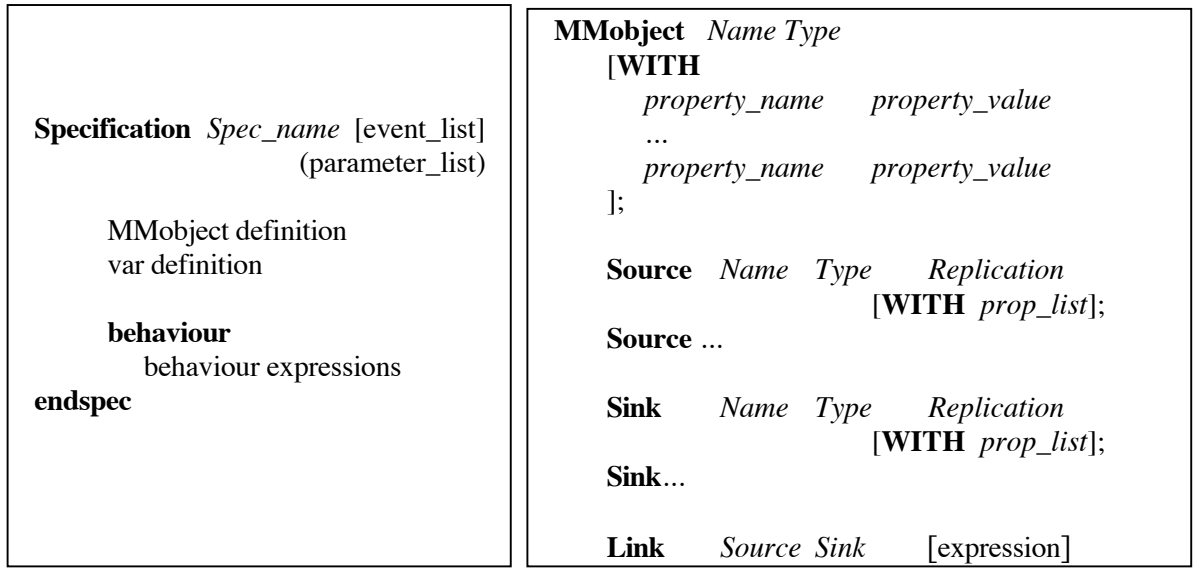

Fig. 4 - Language structure and MMobject declaration structure

The language was inspired on LOTOS. The ADT part was used to describe the type information of MMobjects and components. An MMobject is also a process in the algebra and the behaviour expressions define the rules for synchronizing the applications. It can have a prefix form for a causal style of specification, or use a more structured notation (parallel, enable, etc.) to have more powerful and concise specifications [8]. The ADT part and a set of space names provide a strong type checking environment [9]. The declaration of interest in receiving events is implicit in the language: if the user specifies an expression that includes an event, then there must be a registration to that event.

The topology (2.3.2) is directly described in the MMobject declaration part. It includes the definitions of the "component arrays" used, and their replication degree:

- **Source** name type specifies a "*single*" source
- **Sink** name type [ ] specifies a "*replicated*" sink
- **Sink** name type [ ] [ ] specifies a "*bi-replicated*" sink

The *link* expression specifies:

◊ a list of pairs of components plus ports names (that may be omitted if there is only one port per component); and

◊ an optional expression that can be used to define additional conditions (ex: link to every sink except the one associated with the destination of the source).

The inclusion of port names in the expression is used for "strange" connections, such as connecting the left output channel of a stereo output component to the input channel of a mono input component.

Figures 5a and 5b contains the declaration of MMobjects showed in figures 3a and 3b. The simplicity is notorious.

```
MMObject name type
...
SOURCE source;
SINK   sink[];
LINK   source sink;
END;
```

```
MMObject name[] type
...
SOURCE source[];
SINK   sink[][];
LINK   source sink;
END;
```

Figure 5 - MMobject declarations for MMobjects represented in figures 3a and 3b.

The declaration of different status interfaces per destination is done by replacing the "name" of the MMobject for "name[ ]". This defines *name[i]* as the status interface associated with user *i*, as a valid MMobject reference in the behaviour declaration part.

3. System architecture

An application for shared presentation of information was developed on DIMPLE. The application has a group of users running their front-end applications concurrently and watching exactly the same output. Therefore, the user abstraction described before is well suited, because of the symmetric relation between each user. The application does not perform some CSCW features such as floor control yet. These tasks would require lower level controllers spread in the relevant MMobjects and parameterized by the specification script. Figure 6 shows the architecture presentation system.

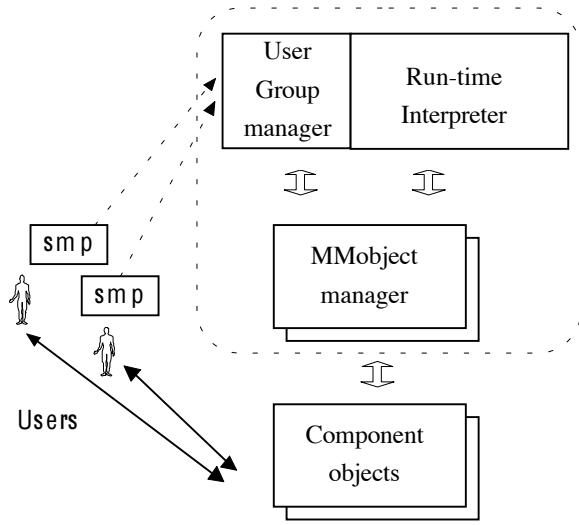

Figure 6 - presentation system architecture

The application is specified by a textual description (a script), written in the language. In this case, the specification is the structure of the multimedia document users can access. Scripts describe the control of the topology and general synchronization of the MMobjects. They are interpreted by the **Run-time Interpreters** (RTI) which builds a state machine from it.

Users control the application via the front-end application, called **shared multimedia presentation** (smp), and interact with the application through the component objects. When a user touches a key, or presses the button mouse (over some image, text or video) an event is generated by the component to every object that registered the interest on it. If the RTI registered the interest on that event (because the script included a transaction associated with the event), then the RTI will receive the event and run the actions specified.

The MMobject interface is controlled by the **MMobject manager** (OM). The OM supports the control features described in 2.3 for the manager, including the management of the components, the topology, and the status interface to the object. The RTI uses one instance of MMobject manager for each MMobject running during the presentation of the application.

The management of the users running an application is done by the **User Group Manager** (UGM). It handles all the data manipulation when users join and leave. The RTI has a *Life status* exactly the same

as for MMobjects (see below). Each time a new user wants to join the system (s)he runs smp with the identification of the application script and a *run keyword*. smp calls the *Prepare* of the RTI with these parameters. As for any other MMobject, one of the results of the *Prepare* is the return of a controlling *Context status* reference. When the user wants to leave smp calls *Destroy* on its *Context* interface. The UGM traps all the *Prepare* and *Destroy* operations over the RTI. The first user that runs the application starts a new instance of the RTI for that particular state machine, and all the MMobject managers needed are also instantiated. For the other users that join afterwards (same application identification and *run keyword*), the UGM only calls the operations *add_destination* over all the MMobject managers *topology status*. The RTI is not aware of the existence of a new user in the system because everything is the same in terms of general synchronizing control. Similarly, the state machine RTI instance and the corresponding MMobjects are only destroyed when the last user leaves the application. For the others that left before, the UGM only calls the operation *drop_destination* over all the MMobject managers *topology* status. In this way, the RTI sees a static interface to the MMobjects when users join and leave an application. At the moment a user joins an application, (s)he sees the same as the others that were already running it, and continues running it in parallel afterwards. Similarly, when a user leaves an application, the others continue running the application, unaffected.

In DIMPLE, an application is just an instance of an MMobject of the type "Controller" (instead of audio, etc.). These MMobjects have a different internal structure (are supported by the RTI), but they offer the same interface as any other MMobject. They have a control interface composed of status, are created through the invocation of the operation *Prepare* over the *Life* status interface of a template server, and support the *Context* status. They start running when the *Play* action of the *Context* status is invoked, and are stopped when the *Stop* action is invoked. The application objects also support other status, such as *Pause*. When an smp joins the application a private *Context status* is given to it. This is transparent to the RTI that thinks it has only one *Context status*. Each operation invocation on a private *Context status* (except *Destroy*) is not trapped by the UGM and has the default behaviour of invoking the operation over all the active MMobjects. This means, for instance, that every user can start or stop the application indistinguishably. As applications are just MMobjects they can be used as simply another object to build new higher-level applications.

4. Shared Access to an Information Retrieval System

4.1. Overview

This section describes in more detail some aspects of the application. The application can work alone or in parallel with the Berkom videoconference system [1] to have audio-visual connection between users.

DIMPLE is implemented on Unix platforms running SunOS4.1.3 and ANSAware, using ATM as the network technology. As already said, control interactions are performed using ANSAware because of its support for distribution and transparencies: ANSAware includes a name server, the trader, that allows a simple but powerful selection to the modules running in the system. The figure 7 shows the modules used when running an application.

The heart of the system was concentrated in a single capsule, the **sserver** (script server), that includes the RTI, a MMobject manager template server, and the UGM. This implementation choice reduced the infrastructure overhead to the minimum possible. It is a rather centralized type of control, and the concentration of all the MMobject managers in a single capsule is intended to minimize the communication overhead between the RTI, the OMs and the UGM. In a previous project where the object model has already been used a distributed type of control was tested [3]. The work showed that the resulting system can be complex. Namely, the delivery of the control messages (events) required the use of atomic multicast protocols to insure the synchronization of the several communicating state

machines. Nevertheless, when specifications start to use other specifications as objects, control ceases to be centralized. There is, however, a master-slave relation not present in the pure distributed case.

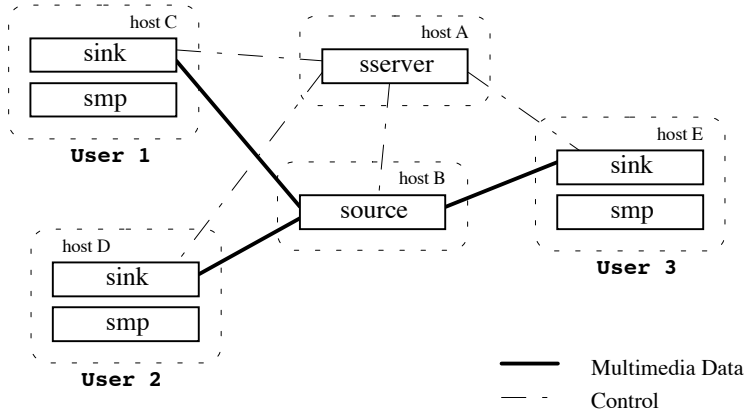

Figure 7 - modules used when running an application.

sservers keep a local application script database, that contains all the scripts supported by that sserver. In a system, there can be several sservers running. The trader is used for the selection of the sservers, and in the current version the "hostname" helps distinguishing amongst the active. A running multimedia application is completely identified by its sserver hostname, the script identification, and the *run keyword*. If two users select the same application identifiers, then they will share the application.

Each smp creates a window control box, that have push buttons associated with the operations in *Context* status (*Play* the application, *Stop* operation, *Quit* the application), and the *Pause* button. It is through this window that the user gives the start and stop commands. When the smp starts, it reads a local configuration file to start the local component template servers (by default it is called '.mmobj_default' but can be parameterized at command line). During the session, it continuously monitors them using a watchdog thread. If a local server dies, then smp automatically exits from the application, and stop every capsule launch by it.

4.2. Multimedia Channels and Interworking with Berkom-MMC

The *port status* was already described. The currently implemented ones are:

- *ansaport* -- an ordinary ANSAware control interface compatible with *port status* which uses RPC-like protocols with implicit binding;
- *udpport* -- a direct use of UDP sockets with a light level for extra support (see below), and with a *connectUDP* operation to perform explicit binding;
- *tcpport* -- a direct use of TCP sockets with a light level for extra support (see below), and with *connectTCP* operation for explicit binding;
- *mmcport* -- an access to Berkom-MMC (see below).

The extra support consists of: (a) a rate control feature taking into account real-time, based upon the use of synchronous up-calls routines to the component objects; (b) jitter compensation algorithms (the used in the current version are very simple, but a new version is now under development and will be reported in the near future. The algorithms are based on the controlled drop of multimedia frames, when their deadline times cannot be met; (c) as described above, the port control interface is used to manage multipoint connections (using the *Connect* and *Disconnect* operations) as sets of 1:1 connections. This implementation choice makes it ready for a clever use any multicast features of the ATM networks in the future.

The MMobject manager is also responsible for the management of communication with alien system components. This is the case of Berkom-MMC. DIMPLE and MMC had to interwork because the audio device of the workstations cannot be shared by user applications (the video board can and each system competes for it). Fortunately, the control features of the audio MMobject are concentrated on the source component, so a simple connection to MMC sinks is enough.

MMC provides an API to access its audio sink, using the OSI stack (ISODE) for control of the MMC audio transport channels. The solution found was to create another *port status* type, *mmcport*, and its *Connect* operation simply instructs MMC API to connect the source component (DIMPLE) to the sink (MMC). mmcport status makes usage of a new type of port address reference, based exclusively on the hostname (used internally by the Berkom system), instead of ANSAware references. Additionally, a new syntax was defined to specify the connection between a mmcport and a "host" (represented by a '*#*'). In terms of MMobject control nothing changes. In terms of topology control the orders are slightly different in one case and in the other. Each component type has an access policy that defines how components of that type must be accessed (what type of the port references must be used during connect, etc.). The MMobject manager uses this information during the management of the components, and the linking of that component.

4.3. Existing Objects

In the current state of development, the system includes some basic MMobjects: video; audio, image and text. Other MMobjects can be integrated without disturbing the existent objects or applications.

The video object uses moving JPEG encoding and provides an interface to the XVideo Parallax board, that does hardware JPEG compression and decompression. It uses two components: mjpegsrc (moving jpeg source) and mjpegsnk (moving jpeg sink). They support the status *Annotation* (can use intermediate positions on the video for synchronization purposes), *Pause* (can be paused), *Speed* (can change the frame rate, or present the video backwards). For transferring the data, it can use TCP or UDP protocol. The first one allows a greater frame rate in a LAN environment. However, if TCP is used on a WAN, with higher error rates, it would be worse than UDP because of the retransmission algorithm.

The image object provides the presentation of sequences of images of several formats (GIF, PCX, Sun Rasterfile, etc.). It uses two components: imagsrc (image source) and imagsnk (image sink). They support the status: *Click* (can generate an event when click by the user), *Annotation*, *Pause* and *Speed*. For transferring the data it currently uses the *ansaport status*. The image object emulates the semantics of a button.

The text object uses a markup language for font and sensitive word definitions. It uses two components: textsrc (text source) and textsnk (text sink). They support the status *Click* and *Annotation*. To transfer data it currently uses *ansaport status*.

The audio object offers two different objects to DIMPLE (two *Life status*). The first one, the "audio" object, is used when the system is running alone, and has two components: audiosrc (audio source) and audiosnk (audio sink). It uses *udpport status* transfer data. The second one, the "audiommc" object, works in parallel with Berkom-MMC, and uses one component: audiommcsrc (audio MMC source). They support the status *Annotation*, *Pause* and *Volume* (to change the volume of the audio).

4.4. Example application

The application, showed in figure 8, consists of an access to a document describing the DIMPLE platform with some example objects. Its script name is Dimple. When a user wants to run it in a shared mode with another user who is already running it with the key learnDimple, (s)he must start smp with the identification ("smp Dimple learnDimple"). The control box appears and the new user starts to watch the same images as the other user was watching. The document consists of a first index page about DIMPLE with sensitive words to start subdocuments. The *mjpeg* word was clicked and an image object with buttons on it shows the various features of the video object. One of the users clicked on the button *Play* and a short movie about a castle is shown.

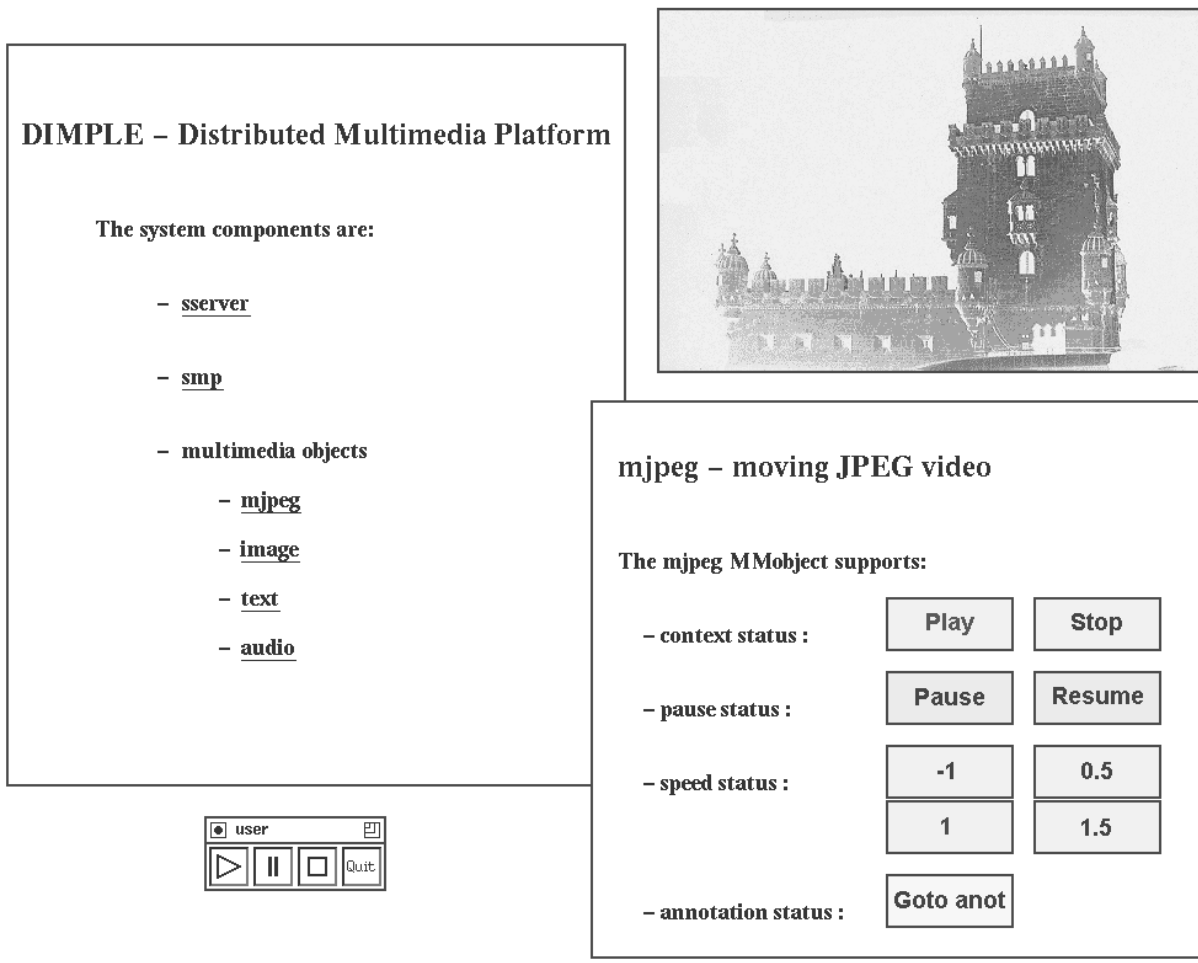

Figure 8 - Example application

5. Related work

Several works describe experiences covering certain parts of what was discussed in this paper. The ones which overlap the most are: [7] where the general concept of a multimedia socket and plug is defined with a reduced set of actions and some type compatibility rules; [5] contains the five ODP viewpoint specifications of what a stream binding object should be (for multimedia connection). However, their choice of TCP for data transportation can prevent any practical application due to the retransmission algorithm. The duplex nature of the streams is not necessary in most of the applications.

[4] presents a general taxonomy for the CSCW in open distributed systems but does not cover continuous media very deeply.

6. Conclusions and Further Work

This paper has shown how an application for multimedia cooperative work can be easily constructed with the concept of distributed objects and encapsulation of the control of data. The DIMPLE platform works at control level which is the level required for most of the general purpose multimedia applications. It is difficult, of course, to express data manipulation at this level (e.g. particular algorithms for mixing signals) but the simplicity gained in constructing other types of applications overcome this limitation.

The separation of control and data allows for the integration of other systems in a very easy way. This is particularly interesting when general purpose networked devices will be standardized in the near future (a kind of X for video and audio). The DIMPLE platform will interwork with them in a very straightforward way using the abstraction of port.

An important aspect which enriches the platform is the connection controlling activities inside the MMobjects. The Reference Model for ODP has a concept for it -- Binding Object. If these activities can be controlled in a simple way from outside, and can interwork between each other in controlling terms, then the specification of an application would be very simple. An example is the association of a floor control algorithm with the enlargement of the video window and the rise of the audio volume for the speaker. The specification would simply provide the policies for that. Some more work on a clear definition of a proper interface to these type of facilities is important.

In terms of communication protocols it was felt the need for multimedia transport protocols and clever usage of the transmission medium by the port part of the objects. The high-performance connection of the port stubs to the hardware devices is also a critical point in the overall efficiency.

Some issues are still open for further research. One of them is the interface to the controlling facilities of the port part. A related one is the definition of high-level QoS parameters for the data streams. Another open issue is the definition of a user “role” semantics, that allows the definition of applications with several kinds of users (e.g. a game with the players, and the viewers).

In terms of real-time protocols, a low-level synchronization algorithm to enforce intra-stream synchronization and inter-stream synchronization between different media in an MMobject is currently being developed and will be reported in the near future. The current real-time enforcing algorithm has no control on the continuity of data.

A final point is the definition of a syntax to express complex topologies in the language, as it was explained in the text.

Acknowledgments

The authors wish to acknowledge Companhia Portuguesa Rádio Marconi for the partial funding of this research. Part of this work was performed under the scope of an EURESCOM project called EMMA - *European Multimedia Experiments in an ATM Environment*.

References

[1] M. Altenhofen, J. Dittrich, R. Hammerschmidt, T. Käppner, C. Kruschel, A. Kückes, T. Steinig, *"The BERKOM Multimedia Collaboration Service*", ACM Multimedia 93, pp. 457-463.

[2] ANSAware 4.1, System Programming in ANSAware. Document RM.101.02, February 1993.

[3] L. Bernardo, "*Specification and Synchronization of Multimedia Aplications with Distributed Control*", Msc Thesis, Instituto Superior Técnico, Lisboa, Portugal, June 1994 (in Portuguese).

[4] G. Blair and T. Rodden, "*The Challenges of CSCW for Open Distributed Processing*", International Conference on Open Distributed Processing 1993, pp. 99-112.

[5] V. Gay, P. Leydekkers and R. Veld, "*Specification of Audio/Video Exchange Based on the Reference Model of ODP*", Proceedings BRIS'94, pp. 179-191.

[6] ISO/IEC 10746-3, ITU-T Rec. X.903, Open Distributed Processing Reference Model - Part 3: Architecture, 1995.

[7] C. Nicolaou, "*An architecture for real-time multimedia communication system*", IEEE Journal on Selected Areas in Communication, 8(3), pp. 391-400 (1990).

[8] P. F. Pinto, P. F. Linington, "*A language for the specification of interactive and distributed multimedia applications*", International Conference on Open Distributed Processing 1993, pp. 217-234.

[9] P. Pinto, L. Bernardo, P. Pereira, *"A Constructive Type Schema for Distributed Multimedia Applications*", Proceedings BRIS'94, pp. 419-434.