WWW-enabled delivery of Interactive Multimedia Documents

Th. Markousis, D. Tsirikos
Dept. of Informatics, University of Athens, Greece

M. Vazirgiannis
Dept. of Informatics, Athens University of Economics & Business, Greece

Y. Stavrakas
Dept of El. & Comp. Engineering, National Technical University of Athens, Greece

Abstract
The structure and delivery of Interactive Multimedia Documents (IMDs) has been an issue of intensive research in the last years. In this paper we propose a scenario based structure for IMDS and a mechanism for WWW-enabled delivery of such IMDs. The structure for the IMDs, based on the concept of scenarios, which define the flow of the presentation and the possible user interactions with the IMD. The delivery mechanism consists in a client/server system, which supports the remote presentation of IMDs. The whole framework has been implemented in Java using the Remote Method Invocation (RMI) client server communication protocol and the Java Media Framework (JMF) for handling multimedia objects. The system presents a promising approach for distributed interactive multimedia on the Internet and Intranets.

Keywords
Interactive Multimedia, WWW, Distributed Systems, Multimedia Synchronization, Java.

1 Introduction

Although multimedia has been with us for a while, the rapid expansion of the Internet, in conjunction with the ever-increasing capabilities of personal computers, have been the driving factors behind the proliferation of distributed multimedia. The media objects do not have to be replicated locally in order to be presented; they can be viewed on demand directly from the site where they reside. With technologies such as xDSL and cable modems, which are set to remove the bandwidth bottleneck at the local loop, the last barrier for the unanimous use of distributed multimedia will be removed. Unfortunately, multimedia document standards such as HyTime [Buf96] or MHEG [ISO93] offer limited interactivity.

On the other hand, authoring tools (e.g. Asymetrix Toolbook, Macromedia Director, Authorware) perform better in that area, especially newer versions such as Toolbook II, which is primarily intended for web deployment. But such tools approach the issue from a different perspective, focusing primarily on company-wide training and trading generality for ease of use.

The issue of distributed Interactive Multimedia Documents (IMDs) and specifically their retrieval and execution is an issue of current research [Kar96][Hua97]. However, most of the proposed systems suffer from limited interaction support both at the modeling and at the implementation level. Hereby we have to stress the importance of interaction as means of document representation. Moreover the delivery of interactive multimedia content through the WWW is an upcoming requirement due to the increasing quantity and quality of such content that enterprises and educational institutions want to disseminate.

This paper describes a Java-based client-server system for WWW enabled delivery of IMDs supporting a high level of interactivity and distribution of scenario and media. In the design of the proposed system we adopted a rich IMD model [VB97][VTS98] that widely covers important features such as interaction and spatiotemporal synchronization. According to the model an IMD is defined in terms of *actors*, *events* and *scenario tuples*. Actors represent the participating media objects (video, sound, image, text, labels and buttons) and their spatiotemporal transformations (i.e. width, height, position on screen, volume etc.). An actor does not contain the media to be presented, but uses a pointer to the location of the media. Events are the interaction primitives and they may be atomic or complex. They are generated

by user actions, actors state changes, or by the system. In [VB97] the reader may find details on a rich model for events in IMDs. The primary constituent of an IMD is the scenario, namely a set of scenario tuples. A tuple is a fundamental entity of functionality in the scenario, conveying information on which event(s) start (or stop) a set of synchronized media presentations (called *instruction streams* in our model). Instruction streams are expressions that involve Temporal Access Control (TAC) actions such as start, stop, pause, and resume, on actors with the use of vacant temporal intervals. In [VTS98] a set of operators has been defined for the TAC actions and for the corresponding events. The operators are: >, !, || and |> corresponding to the actions start, stop, pause and resume respectively. Thus the instruction stream: (A>3B>0C!) is interpreted as "start A, after 3 seconds start B and immediately after that (0 seconds) stop C".

The rest of the paper is organized as follows: In section two we refer to related work in section three we refer to the technical choices we did in terms of software tools and platforms, while in sections four and five we elaborate on the design and implementation of the server and the client respectively. In the last section we summarize our contribution and indicate directions for further research.

2 Related Work

Substantial research has been conducted in the field of multimedia storage servers, especially video servers. Until recently, the standard way to present multimedia through a network was to download them first, and then display them. However, for average to large media object sizes, this solution is far from optimal. This is due to reasons such as start-up latency usually being unacceptable, having considerable storage requirements at the client side, and that there is no way for a user to preview a video in order to decide whether he or she really wants to see it. For real-time applications, there is a strong need for streaming media, that is, for media that are presented while they are being downloaded and are not stored locally. Most of the work in this field has been carried out by the industry. Some related efforts are the following:

- RTSP, developed by Progressive Networks and Netscape Communications [Sch97] to cover the need of transferring multimedia through IP networks. RTSP offers an extensible framework for transferring real-time data such as audio and video. It has been designed to work with established protocols such as RTP and HTTP and constitutes a complete system for the transmission of streaming media through the Internet. Its advantages, as pointed out by its designers, include i. High reliability over the existing network infrastructure. ii. Reduced overhead during data transmission. iii. Security. iv. Multiple platform support (Mac, Windows 95/NT/3.1 for the client, Mac, Windows NT, Unix for the server). v. Acceptance and support from the industry. Moreover, RTSP has been submitted to the Internet Engineering Task Force (IETF) as a standard protocol for multimedia streaming at the Internet. Despite its series of advantages, RTSP has some drawbacks: its specifications, as well as its implementation, are still at a very early (alpha) stage. In addition, it is written in C, requiring different executables for each platform, and its inclusion in the application code is far from plug-and-play.
- Microsoft Netshow. It's Microsoft's product for the transmission of streaming multimedia through Intranets, LANs and the Internet. It supports multicasting and unicasting, and it can broadcast stored data, as well as live feed. It consists of a server which executes under Windows NT, a set of administrative tools, and a set of client programs that operate autonomously or in co-operation with other programs. There are currently clients for Windows 95/NT, while clients for Mac and UNIX are being developed. Some tools exist for the attachment of live audio and video to streaming format with the addition of a URL and optionally of instructions to the client. NetShow is a relative technology to ActiveMovie, which is used by the Intel implementation of the JMF.
- Berkeley Continuous Media Toolkit (CMT) [Smi96]. It is a framework that consists of a suite of customizable applications that handle streaming multimedia data. It requires

significant programming to support each medium. On this toolkit, cmplayer [Pat96] was built, an application that is used to present remote continuous media in conjunction with a web browser. It supports the synchronized reproduction of multiple media and is available for most platforms.

All the above projects concentrate on how media are transferred and displayed, but the context in which the media are to be presented is not considered. This is the field of distributed multimedia applications, which has received less attention by researchers. Some of the most interesting efforts are:

- NetMedia. A client-server architecture that can be used for the development of multimedia applications. It is described in [Jon97], where algorithms are described for the effective synchronization between media and the maintenance of a QoS. It is assumed that the server and the network offer adequate performance regarding the delivery of media objects. The paper defines a multimedia presentation as a set of media streams, thus making NetMedia similar to the approaches described above.
- CHIMP. Here the definition of multimedia document is broader than in other efforts, since such a document "consists of different media objects that are to be sequenced and presented according to temporal and spatial specifications" [Can96]. CHIMP focuses both on the authoring and the presentation of documents, the latter being driven by a presentation schedule that describes the starting times and duration of the various media. The temporal constraints can be quite complex and powerful, but the lack of support for user interaction limits the system to pre-orchestrated presentations.
- The system proposed in [Man97] involves more user participation, since it's the user who controls the course of the presentation. A framework for link management within hypermedia documents is described, which supports embedding dynamic links within continuous media such as video, as well making queries to a database. There is a prototype implementation for UNIX, which uses NFS to fetch the media, practically limiting it to LANs only.
- A similar architecture is presented in [Ma96], which suggests the use of video-based hypermedia to support education on demand. The system is based on URLs, which are embedded in QuickTime movies. There is no notion of time or events, and the systems bias towards applications such as video-based lectures could be the reason for the limited interaction it offers.
- The time factor is systematically considered in [Nan97], where temporal as well as spatial events between multiple media are defined. A prototype scenario-authoring tool based on Windows 95 is described, while the scenarios it produces can also be reproduced in different platforms. Despite the extensive coverage of spatiotemporal events, the lack of support of any user interaction puts the system under a different perspective.

In the context of WWW-enabled delivery of IMDs, special attention should be paid to the upcoming standard SMIL. The key to HTML success was that attractive hypertext content could be created without requiring a sophisticated authoring tool. Synchronized Multimedia Integration Language (SMIL)[SMI98] aims at the same objective for synchronized hypermedia. It is an upcoming standard for synchronized multimedia to be presented in a WWW browser. SMIL allows integration of a set of independent multimedia objects into a synchronized multimedia presentation. A typical SMIL presentation has the following features:

- the presentation is composed of several components that are accessible via a URL, e.g. files stored on a Web server.
- the components are of different media types, such as audio, video, image or text.
- interaction support in terms of simple events. This implies that the begin and end times of different components have to be synchronized with events produced by internal objects or by external user interaction. Also simple user interaction is supported. The user can control the presentation by using control buttons known from video-recorders, such as

stop, fast-forward and rewind. Additional functions are "random access", i.e. the presentation can be started anywhere, and "slow motion", i.e. the presentation is played slower than at its original speed.

- support for hyperlinks, the user can follow hyper-links embedded in the presentation
- rich high level temporal composition features such as lip-synchronization, expressive temporal relations (including parallel with a master, interruptions with the first ending element (par-min), wall-clocks, unknown durations, etc...

SMIL in its current form does not support relative spatial positioning and spatial embedding of media objects, an aspect extensively covered by our authoring approach. Moreover, the temporal synchronization model does not address the causality of temporal relationships, while the interaction model is rather limited as regards the multitude of events that may occur in a presentation. The temporal composition is represented as a subset of the Allen relations (essentially the authors deal with sequential and co-start /co-end and equality in time relationships). Then a temporal graph is constructed and the global set of constraints is solved offering alternative durations for the objects and the whole presentation.

The concluding remark here is that there is an extensive coverage of topics such as streaming video, as well as multimedia presentations. However, the merging of the two areas combined with substantial user interaction has received little attention by researchers.

3 Architecture and Implementation

The system implementation is based on Java and other accompanying technologies due to its appealing features, such as built-in multi-thread support and cross-platform compatibility. Moreover, all major WWW browsers support Java, thus making the presentation of an IMD feasible in any WWW browser.

Both the client and the server are implemented in Java and are, therefore, portable through a variety of platforms. The continuous media, video and sound, are retrieved by the client from http servers and are presented with the aid of the Java Media Framework (JMF) [Jav97-2], which specifies a unified architecture, messaging protocol and programming interface for media players, media capture, and conferencing. JMF APIs support the synchronization, control, processing, and presentation of compressed streaming and stored time-based media, including video and audio. Streaming media is supported, where videos and sound are reproduced while they are being downloaded without being stored locally. This feature is extensively exploited in the system described here.

It is evident that the choice of Java as the main implementation platform has some disadvantages, mainly related to performance. Java has often been criticised of low performance. This potential problem is circumvented by the use of JMF to view sound and video (the media with the largest amount of data to process), which in turn uses native libraries (as opposed to Java code) for the actual presentation. While testing the system, we observed that the main bottleneck is the network, especially when running clients over WANs.

The communication between the client and the server is performed exclusively (except for the actual streaming of media) using the RMI (Remote Method Invocation) protocol [Jav97-2], a technology used to seamlessly distribute Java objects (in our case IMDs) across the Internet and intranets. The choice of RMI instead of CORBA for the communication protocol is based on the following reasons:

- RMI allows Java objects to call methods of other Java objects running in different virtual machines. These methods can return primitive types and object references, but they can also return objects by-value.

- RMI has a very flexible way of providing support for persistent objects (the IMDs in our case) as it uses Java serialisation to store and retrieve objects. Serialisation is efficient and easy to use and suits well our case as we only need to store scenarios without any need for complex operations on objects, relations between them or transaction processing.
- Furthermore RMI does not require the use of extra software such as ORBs.

An important architectural choice in our system is the *logical* and *physical* separation of the content (media data) and structure of a scenario. The advantages of the logical separation are well known and are explained in many approaches. The physical separation may contribute to significant enhancement of Quality of Service (QoS). As the media data can reside in different servers, we should be able to dynamically select the server with the least workload and the best network connection, in order to minimise the presentation delay of the scenario. This can prove extremely helpful when dealing with scenarios that contain a large amount of video and sound data (the most demanding media types in terms of network bandwidth). Concerning the implementation status of the proposed architecture, we have a fully functional server and client that have been successfully used to view example scenarios that range from a few buttons to a full-fledged presentation with lots of sound and images and about 20MB of video. All scenarios were tested over Ethernet LAN's and over a WAN. Areas not yet covered are spatial synchronisation relationships and QoS issues.

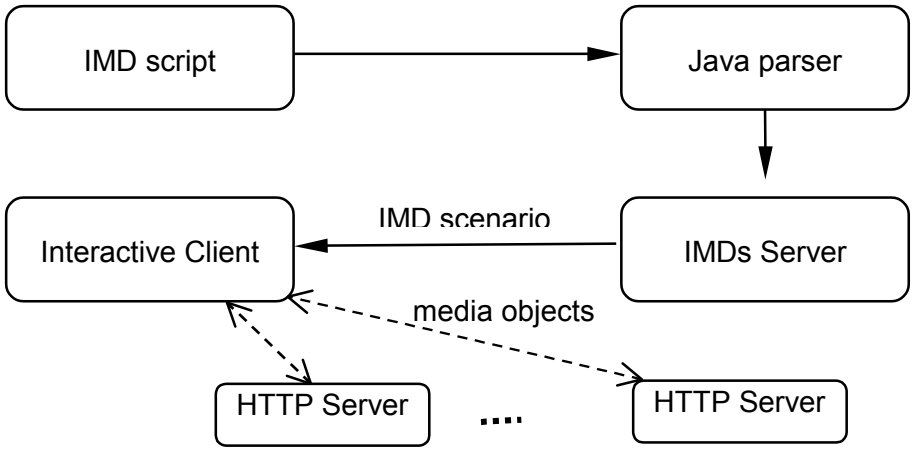

Figure 1. An IMD lifecycle

The lifecycle of an IMD in our system appears in Figure 1. Initially the IMD author creates an IMD script, either using a visual editor developed for this purpose or a text editor, and stores the script in a text file. The format of this file and some examples are given in [VTM+00]. This script (actors, events, scenario tuples) is then parsed and translated into corresponding Java objects. These objects are then stored in a flat file on the server, using the Java Serialization mechanism [Jav97-1]. Then the server is ready to service client requests. Then clients can then request scripts stored on the server using RMI, and receive them as Java objects. Finally, the client presents the scenario to the user; the scenario contains references to audio and video data, which are streamed "just-in-time" from the http servers where they are stored.

4 The Server

The key elements of the system are the IMD objects, containing specifications for actors and the presentation scenario, together with the handling of user and internal interactions. However, an IMD object does not include the media objects themselves, but only references to their locations in the form of URLs. The intent is to minimise download latency and space requirements at the client side, as the media are requested by the client only at the moment

they are to be presented. This depends on user interactions and on the scenario itself (for example if the user presses ButtonA then he will see Image1 whereas if he presses ButtonB he will see Video2). The system architecture is illustrated in Figure 2. The server functionality is provided by three modules. These are:

- The Multimedia Document Server (MDS), responsible for the delivery of IMD objects to the client.
- The RMI registry, which is the naming service of RMI and is used to establish communication between client and server
- The set of http servers on which the media objects reside

The actions that take place during an IMD session are hereafter described. When the MDS is started, it scans a file that contains all IMD objects and stores their names and descriptions. Then, it registers itself to the RMI registry and waits for client requests. When there is a client request for an IMD, the corresponding object is retrieved from the server. During the IMD session, whenever a media object (video, sound, image, text) is to be presented, the client communicates with the respective http server through calls to the JMF (in the case of video and sound) and presents the media object directly from the remote machine without storing it locally.

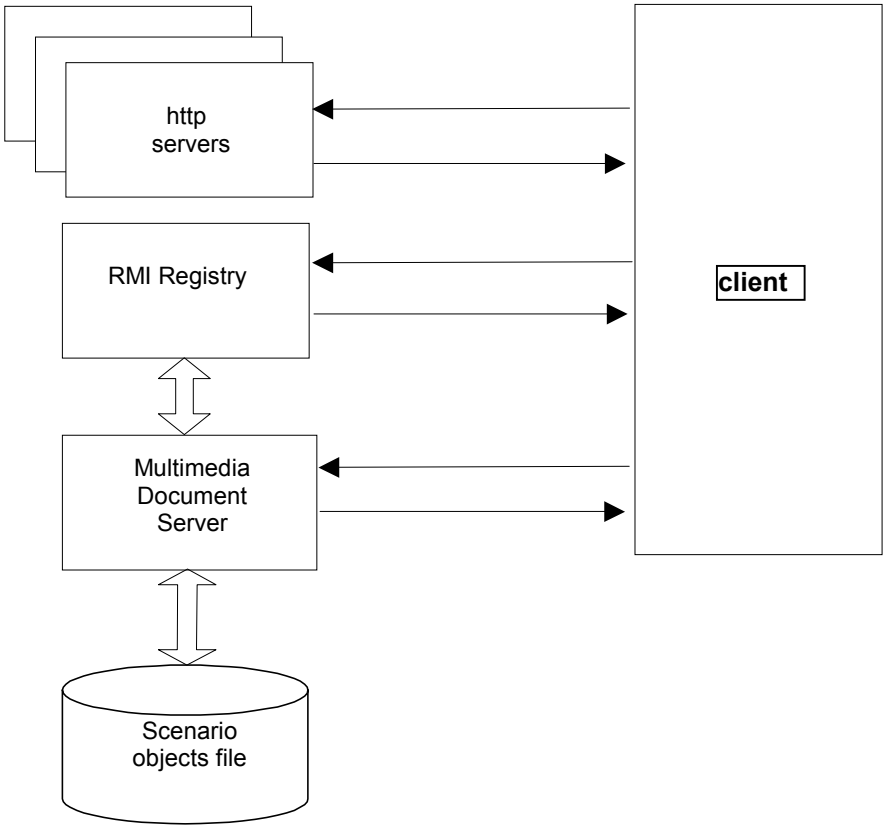

Figure 2. The server architecture and the communication with the client

The separation of the scenarios from the media enhances the flexibility of the system, while giving greater control over a potential layer for QoS maintenance. The high availability of http servers can be used for massive distribution and replication of media, thereby reducing load on any individual server, whereas one server may serve the requests for scenarios (time-independent and usually a few kilobytes in size). This scheme guarantees high scalability for the server system, since more http servers can be added to accommodate increased load. The only practical bottleneck can be the MDS, but only after the number of concurrent clients increases very significantly.

5 The Client

The client retrieves scenarios from the server, media objects from the appropriate http servers, and presents them appropriately. Before describing in detail the client architecture we will briefly present the basic elements of the scenario tuple model[VB97][VTS98]. The term scenario in the context of IMDs stands for the integrated behavioral contents of the IMD, i.e. what kind of events the IMD will consume and what actions will be triggered as a result. The scenario, in the current approach, consists of a set of autonomous functional units (*scenario tuples*) that include the triggering events (for starting and stopping the scenario tuple), the presentation actions to be carried out in the context for the scenario tuple, related synchronization events and possible constraints. More specifically a scenario tuple has the following attributes:

- Start_event: represents the event expression that triggers the execution of the actions described in the Action_list.
- Stop_event: represents the event expression that terminates the execution of this tuple (i.e. the execution of the actions described in the Action_List before its expected termination.
- Action_List: represents the list of synchronized media presentation actions that will take place when this scenario tuple becomes activated. The expressions included in this attribute are in terms of compositions as described in previous sections and in [VTS98].
- Synch_events: refers to the events generated (if any) at the beginning and at the end of the current tuple execution. These events may be used for synchronization purposes.

An IMD scenario execution scheme must have the following features: ability to detect and evaluate events generated by the system, the user or the actors; maintenance of information about the past and the present status of the IMD session. Such information should include presentation actions along with a timestamp, the events generated by these actions, a history of all the occurring events, the active scenario tuples, etc. Other important features of a scenario execution mechanism are: asynchronous activation of tuples, concurrent execution of all instruction streams, and synchronized presentation actions according to the scenario tuples' specifications.

In the sequel we will present the client architecture. Each element of the scenario model (i.e. tuples, events and actors) has been implemented as a different Java class. Therefore, we have a *ScenarioPlayer* class and a *TuplePlayer* class that are capable of handling an IMD scenario and a scenario tuple respectively. *Listener* is in charge of presentation of a single media object and also detecting all events related to the single object presented. Class *AppEvent* stores information on all the events that may occur in the specific IMD session. Another fundamental class of our client design is the *EventEvaluator* class, which is in charge of evaluating the start and stop events of all tuples each time a simple event occurs and sends the appropriate messages to the *ScenarioPlayer.*

The outline of an IMD client functional architecture follows (see also Figure 3). When an IMD session is started, the "StartApp" (start application) event is generated and sent to the *EventEvaluator* that determines which tuple(s) are to be started. The *ScenarioPlayer* then creates the corresponding *TuplePlayer* objects, which in turn create as many *InstructionStreamPlayer* objects as necessary. The latter present media objects according to the scenario by creating the appropriate *Listener* objects. Each *TuplePlayer* and *InstructionStreamPlayer* object created corresponds to a new thread.

During scenario execution all generated events that are of some interest to the IMD are sent to the *EventEvaluator,* which evaluates them and notifies the *ScenarioPlayer* for any changes in tuple states. Then the *ScenarioPlayer* starts/stops the appropriate tuples. When a tuple must be interrupted, all the participating actors are interrupted (if they are active) and the appropriate event is sent back to the *EventEvaluator*.

The *EvaluatorNotifier* objects are in charge of storing events produced by objects presented until the *EventEvaluator* can handle them. When the *EventEvaluator* finishes processing an

event, it notifies all *EvaluatorNotifiers* that may have a new event to send, that it is now available to process a new event. It then receives a new event, falls again into the locked state and processes it. By having the *EventEvaluator* notify the *EvaluatorNotifiers* instead of having them poll the *EventEvaluator* at regular time intervals we have a significant gain in performance as the *EventEvaluator* is used as soon as it is needed.

The client application is in charge of two main tasks: starting and interrupting scenario tuples on the basis of occurring events, and presenting media objects according to the specifications of the instruction streams in the scenario tuples. In the following subsections we present in more details these tasks and the related Java classes of the implemented system.

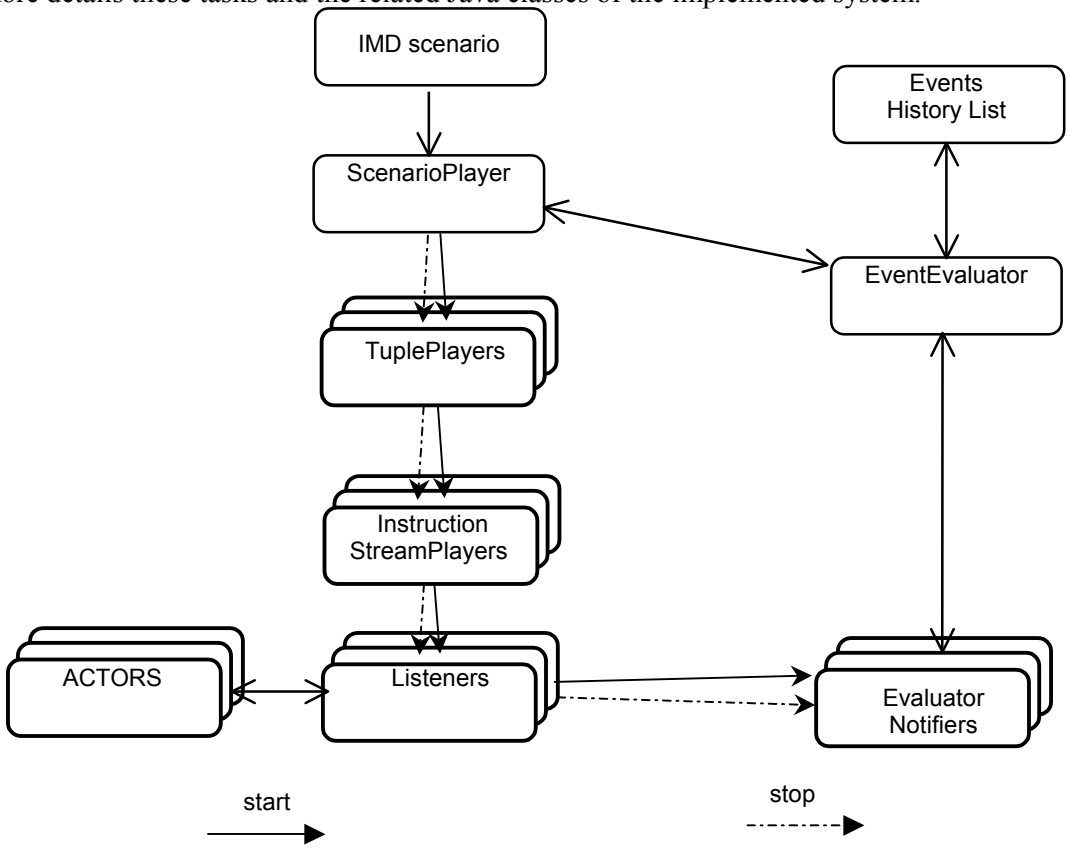

Figure 3. The architecture of the client for IMD scenario execution

5.1 Starting and interrupting scenario tuples

In order to accomplish this task, the client must detect and evaluate the events that occur in an IMD session and match them against the events included in the start/stop event attributes of the scenario tuples. A tuple becomes *active* when the start event of that tuple is evaluated as *true*. At this point all instruction streams of the tuple start execution at the same time, though they do not have to stop concurrently.

To make the concept of a tuple clearer we give an example.

```
Tuple                   :t1
start_event             :e1 AND e2                      (complex event)
stop_event              :e3 OR ANY(2;e4,e5,e6)          (complex event)
instruction_streams     :(button1 > 0 button2 >),
                         (image3 /\ video4),
                         (text2 > 5 image3!)
start_synch_event       :none                           (simple event)
stop_synch_event        :e8                             (simple event)
```

The above tuple implements the following functionality: when the conjunction of events `e1` and `e2` occurs the instruction streams `(button1 > 0 button2 >),(image3 /\ video4), (text2 > 5 image3!)` will be simultaneously started. If during the execution of these instruction streams the complex event `e3 OR ANY(2;e4,e5,e6)` occurs they will be forced to stop and the tuple to fall in the idle state. The end of the scenario tuple generates the event `e8`.

When an IMD session starts, none of its tuples is active. The "StartApp" event is generated and the tuples whose start event is the "StartApp" event start their execution. A tuple cannot be restarted when it is active, even if its start event becomes true. A tuple can only start again once it has stopped/finished and, thus, is in idle state.

Once a tuple has been initiated there are two ways it may end: *forced* or *natural*. In the first case the tuple stops when its stop event becomes true. In the second case the tuple falls into the idle state when all instruction streams have finished. An instruction stream is considered as finished when all the involved actors have fallen into the stopped state. A tuple's life cycle can be represented by the state-diagram shown in Figure 4.

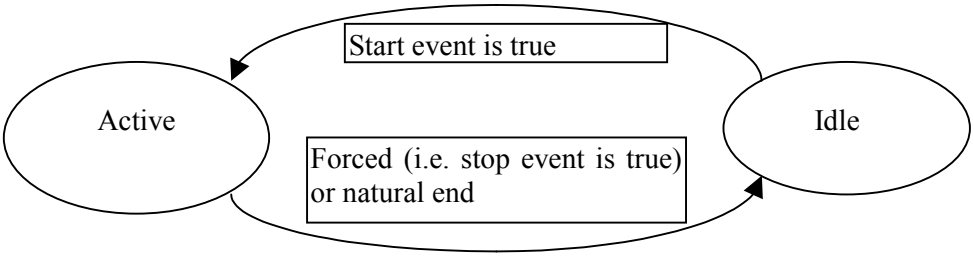

Figure 4. Tuple life cycle

Hereafter, we clarify the semantics of interrupting an actor. For this purpose we distinguish actors with inherent temporal features (sound or video), and actors without such features. An actor of the first category stops either when its natural end comes (there is no more data to be presented), or when it is stopped using the stop operator "!" [VTS98]. Actors of the second category (e.g. an image) stop only when we apply the stop operator on them, and thus no longer wish to have them displayed.

In the sequel we present the classes that are related to event detection and evaluation. However, before proceeding we must distinguish between simple and complex events. Simple events can be generated by:

- an actor changing state (starting, stopping, pausing and resuming),
- the system ("the time is 12:45", or "the QoS is below average") or
- the user ("button1 pushed", "the cursor is over image2").

Complex events are combinations of simple events, using a set of composition operators [VB97] (e.g. "event e1 AND (event e2 before event e3)"). The only complex events are the start/stop events of the tuples.

The *EventEvaluator* class evaluates simple events against the start and stop events of all the scenario tuples. This means that when a new event occurs the start and stop events of the tuples are evaluated, and those that are found to be true trigger the appropriate action(s). This task may be complex since the start/stop events may be arbitrarily complex, so an incremental evaluation as new events occur is necessary. The *EventEvaluator* additionally controls the synchronization of all threads that send messages to it about events that occurred. This function is further explained in the *EvaluatorNotifier* class presented in the next subsection.

As mentioned before, the scenario model provides for complex events evaluation that represent complex interactions. For instance, consider the complex event: SEQ(e1,e2). As it is profound in order to evaluate this expression we need to detect an occurrence of e2 after an occurrence of e1. This implies that the evaluation of complex events requires some kind of "memory" as regards events that have already occurred. This requirement is fulfilled by the *HistoryList* class. There is only one instance of this class in each IMD session, maintaining information on the events that have occurred, from the start to the current time, which is defined as the time elapsed in seconds since the IMD session start. For each event we keep all the timestamps of its occurrences. For example, the entry: <e1, 3, 6, 12> implies that event "e1" occurred 3 times with the timestamps: 3, 6 and 12. It is important to clarify that in the *HistoryList* all simple events and all tuple synchronization events occurring are stored. In this structure there is no information on events that are defined by the scenario author, but have not occurred until the current moment.

When an event occurs, the *EventEvaluator* locks itself (i.e. does not accept any further messages from other classes, *except itself*) until the event evaluation process finishes. It is important to stress that during the period of event processing (*EventEvaluator* in locked state) the occurring events are not lost but are maintained in the respective *EvaluatorNotifier*s. When the *EventEvaluator* finishes processing an event, it notifies all *EvaluatorNotifiers* that may have a new event to send, that it is now available to process a new event. It then receives a new event, falls again into the locked state and processes it. By having the *EventEvaluator* notify the *EvaluatorNotifiers* instead of having them poll the *EventEvaluator* at regular time intervals we have a significant gain in performance as the *EventEvaluator* is used as soon as it is needed.

During this process, the aim is to find all tuples that have to start or stop due to the occurrence of the current event. Once all tuples that must either start or stop have been found, the *EventEvaluator* sends the appropriate messages to the *ScenarioPlayer* to act accordingly (i.e. start or stop the tuple) and continues processing any synchronization events that these tuples may have defined. In some cases an occurring event may cause both the start and the stop event of a tuple to be evaluated as true. In this case, the system allows a tuple to either start or stop, according to its previous state, but not both.

5.2 Presenting the media according to the scenario temporal synchronization relationships.

In this section we present the client's classes that are in charge of presenting the media objects according to the synchronization relationships included in the instruction streams. Not all instruction streams have the same effect on actors. In this respect we distinguish two categories of instruction streams.

The first one includes instruction streams whose synchronization expression starts with an actor and the start operator (>). These instruction streams start immediately after the tuple activation and remain active until all participating actors stop. The second category includes instruction streams that contain the synchronization operator "/\". These instruction streams start just after the activation of the tuple and remain active until the temporally shorter of the involved actors ends its execution. If an instruction stream contains the synchronization operator "/\", it cannot contain any other operator (i.e. >, <, ||, \>).

The *InstructionStreamPlayer* class is designed to execute an instruction stream as defined in earlier sections. The instruction stream string is parsed at execution time. For example, assume the instruction stream: "video1> 4 image1> 0 button1> 5 video1||". It implies that the video clip "video1" should start, and after 4 seconds the image "image1" be presented. Immediately after the button "button1" is presented and after 5 seconds video1 is suspended.

The *InstructionStreamPlayer* will start parsing the instruction stream and will find string "video1". This string must correspond to an existing actor (actually the ScenarioPlayer verifies the name, since this class maintains information about the actors). Once *InstructionStreamPlayer* gets the actor "video1", it continues parsing to find that the start operator (">") is to be applied to video1. This is accomplished by creating a new *VideoListener* object (see further for explanation of the *Listener* class), which starts the video presentation according to the specifications of the corresponding actor. While the *VideoListener* is performing this task, the *InstructionStreamPlayer* continues parsing finding the 4 seconds pause. This is accomplished by falling in the idle state (i.e. release the CPU) for 4 seconds, and then continue parsing. The same steps (find actor, find operator, apply operator to actor, and wait for a number of seconds) are repeated until the whole instruction stream is processed. A new listener object is created only when the start operator is found. For the other operators the *InstructionStreamPlayer* does not create a new *Listener*; instead it sends the appropriate message (pause, resume or stop) to the listener previously created.

In the case of synchronization expressions including the operator "/\" (implying parallel execution, for example assume "video1/\ button1/\ sound2/\ text3"), things are somehow different. All actors participating in the instruction stream are inserted in an array and then the appropriate listeners are created for these actors. Each listener presents one media object, and when the temporally shorter objects finish, the corresponding listener notifies the *InstructionStreamPlayer* which in turn sends messages to all remaining listeners to stop.

The "Listener" classes (one for each different kind of actor) do not only present actors, but also detect ("listen to") any events generated by the actor they are controlling (i.e. media state changes etc.). For instance, the *VideoListener* class can start, stop, pause and resume a video clip and can also detect all kinds of events that are related to the particular video. The video presentation is done according to the specifications in the corresponding actor (i.e. volume intensity, start point, screen coordinates etc). The same applies to the other listeners, namely *SoundListener*, *ImageListener*, *TextListener*, *ButtonListener* and *LabelListener*.

Each listener occurrence is paired with an instance of the *EvaluatorNotifier* class. This class serves as a filtering mechanism that sends to the EventEvaluator only the events of interest to the IMD (i.e. contained in the *Scenario Events* list). When the *EvaluatorNotifier* receives messages denoting actor state change, it checks whether there is a related event defined in the IMD and, if such an event exists, sends it to the *EventEvaluator*. For example, if the *ButtonListener* for button A detects that the button has been pressed, it will send a "ButtonDown" event to the *EvaluatorNotifier*. The *EvaluatorNotifier* checks if an event "ButtonDown" related to button A has been defined by the scenario author. If such an event exists, it will be sent to the *EventEvaluator* together with the occurrence time (timestamp). The *EvaluatorNotifier* class is responsible for performing the filtering of events so that the EventEvaluator does not have to process redundant events.

5.3 Event evaluation

As already mentioned, the interaction in our system is handled in terms of simple and complex events occurring in the IMD context, and are generated by the user the systems or the actors. Hereafter, we describe the event evaluation process during which a simple event that occurs in the IMD session is compared to the events (simple and complex) of interest to the application. On finding event expressions as TRUE, several actions may be taken such as starting/interrupting a tuple, or interrupting the IMD session.
First we shall present the evaluation of a simple event. This task is accomplished by the function evaluate_simple_event() that follows.

```
EventEvaluator "locks" itself after receiving an event
evaluate_simple_event(simple event e) {
EventEvaluator receives e and writes it to HistoryList
```

```
for each tuple t
        if t is idle
        then evaluate (t.start_event, e)
                if t.start_event is true
                then add t in tuples_to_start array.
        else
        if t is active
        then evaluate (t.stop_event, e)
                if the t.stop_event is true
                then add t in tuples_to_stop array
start_tuples(tuples_to_start)
stop_tuples(tuples_to_stop)
}
EventEvaluator “unlocks” itself
```

It is important to stress that during the period of event processing (*EventEvaluator* in locked state) the occurring events are not lost, but are maintained in the respective *EvaluatorNotifier*s. When the *EventEvaluator* is again available, it receives another event and falls into the locked state again and processes it.

The function evaluate(t.start_event, e) carries out the evaluation of the event e against the event expression stored in the start/stop event of the tuple. The resulting value will determine whether the tuple t should start/stop. This function will be presented in more details further in this section. Hereby we present the algorithms for starting/interrupting tuples whose corresponding start/stop events were found to be true:

```
start_tuples(array tuples_to_start) {
for each t in tuples_to_start
        the ScenarioPlayer starts t
        if t.start_synch_event not null
                then evaluate_simple_event(t.start_synch_event)
}
stop_tuples(array tuples_to_stop) {
for each t in tuples_to_start
        the ScenarioPlayer to stops t
        if t.stop_synch_event not null
                then evaluate(t.stop_synch_event)
}
```

In the sequel we will refer to the way our system evaluates a simple event against event expressions that are stored in the start/stop event of tuples. This task is carried out by the evaluate() method. When the start/stop event is a simple event, the evaluation is limited in searching in the HistoryList for an occurrence of such an event. In the case that the start/stop event is complex, it may contain expressions including operators and functions that are defined in the framework presented in [VB97]. Namely the system implemented the following subset of operators: AND, OR, NOT, and the functions are: ANY, ANYNEW, IN, TIMES, SEQ and (event1: time_indication : event2).

In this case the evaluation process is more complex and is carried out in three distinct steps.

1) The first step is the transformation of the event expression into postfix form resulting in an expression without brackets.
2) The second step is the evaluation of each function appearing in the expression, and replacing the function with the token “true” or “false” if this is the case.
3) The last step is to evaluate the result that now consists of the tokens “true” or “false” combined with the Boolean operators.

The above-described steps are demonstrated in the following example. Assume the event: "e1 AND ANY (2;e1;e2;e3) OR e4" and the *HistoryList* appearing in Table 1:

Event	timestamps
e1	2, 7, 12
e3	5

Table 1: The contents of the History list at time 13 sec.

The *first step* will result in the transformation of the event into the expression (the symbol "/" is the delimiter):

e1/ANY(2;e1;e2;e3)/AND/e4/OR

The *second step* will produce the expression

"true/true/AND/false/OR"

The *third step* will evaluate this expression to "true". If this event is the start_event of a tuple, and the tuple is idle (i.e. has not already started or has already finished), then it must be started.

What would happen if we had multiple occurrences of start/stop events? We dealt with this problem and we propose a mechanism that enables multiple execution of tuples in the same IMD session. Assume tuple t1, t1.start_event = "e2 AND e3" and that the event e1 occurs while with the HistoryList is as appears in Table 2.

Event	**timestamps**
e2	3, 9
e3	7

Table 2: The contents of the History list at time 13 sec.

During the evaluation of e1, the expression "e2 AND e3" would evaluate to true, and provided that tuple t1 had finished playing, we would have to start it again. It is clear that event e1, which is under evaluation, is irrelevant to tuple t1 and it would make no sense to start tuple t1 because of an irrelevant event.

Therefore we have expanded the evaluation mechanism, to check whether the simple event (for instance e1) we are currently processing is related to the tuple whose start or stop event we are evaluating. If e1 participates in the start or stop event expression, then the evaluation expression goes on. Otherwise, the evaluation stops. For instance, in the above example, e1 does not participate in the expression "e2 AND e3", thus the expression will not be further evaluated. This mechanism enables multiple executions of tuples only when the proper events occur.

An IMD session is an environment that involves concurrent execution of several tasks, such as event detection and evaluation, starting, interrupting and monitoring scenario tuples, presenting media objects according to synchronization specifications, etc. As already mentioned we have used the Java's support for threads. Thus, each instance of the classes *Listener, EvaluatorNotifier, InstructionStreamPlayer, TuplePlayer* runs as a separate Java thread. This choice bears some advantages. It enables detection and evaluation of all the events that occur in an IMD session, which otherwise would be problematic as it has been proved in single threaded architectures [Sta97]. Moreover in the case that a media object fails for some reason (e.g. due to network problems) this will affect the rest of the session to a

minimum degree, while in a single threaded approach the application would probably collapse.

6 Conclusions

We have presented a Java-based client-server system for IMDs supporting a high level of interactivity and distribution of scenario and media. The salient features of the system presented are:

- *Platform independence.* The physical separation of IMD structure (scenario) and content (media), allows the usage of external resources for storage and presentation, and reduces the workload and maintenance for the server. The choice of Java as the implementation platform and the storage of media objects in http servers, makes the design appealing for wide Internet usage.
- *Generic multi-threaded approach for rendering interactive scenarios*. We have developed a robust mechanism for detection and evaluation of events as carriers of interaction, and the corresponding synchronized media presentation algorithms. This approach is generic and may be considered as rendering architecture in other emerging application domains like synthetic 3D worlds etc.
- The choice of Java (along with accompanying technologies like RMI and JMF) as the implementation tool and the storage of media objects in http servers makes the design appealing for wide Internet usage. The clients are capable of presenting multiple scenarios simultaneously. The transformation of the IMDs, as defined in our system, into Java applets makes the documents executable in any WWW browser.
- *Ease of use*. It is very easy for someone to create a new scenario. All they need to have is a text editor to write the scenario, according to a BNF grammar we have defined that expresses the theoretical model. The rest is done automatically (i.e. a parser generates a Java object from the text and the author only has to send this object to the IMD server).

The architecture presented here may be extended towards the following directions:

- *More functionality at the server side.* With the present architecture, the client receives an IMD object from the server, which includes the specifications of a multimedia document (actors, events, and scenario). Apart from these attributes, this object could also include methods related to the presentation of the scenario, which are currently included within the client. This would enable the transparent upgrade of the client functions, an upgrade performed at the server side in a centralized way.
- *Distributed IMD objects.* Under the current design, the media data can be distributed over several http servers. The obvious next step would be to extend this policy to the scenario objects. Although this would not bear significant performance improvements, due to the small size and the time independence of the scenarios, it would improve upon the fault-tolerance of the system.
- *Provision of QoS*. Provisions could be made to ensure the QoS. Admission control could be the first step towards this goal. It would be difficult to apply it to scenario requests, since, due to the interactive nature of scenarios, the bandwidth demands of a scenario are not fixed – they depend on user actions, but admission control could be used at every http server when a media object is requested. Moreover, the client could monitor the playback performance (or test it before the actual media object presentation), and refuse to present the video or sound if quality falls below a certain threshold. It is clear though, that due to the massively distributed architecture of the system, there is no apparent way of applying a centralized QoS control. In its present state, the system operates on a best-effort basis.
- *Database support at the sever side*. Another extension would be the storage of IMDs in a database system. This will make the server capable to serve large quantities of IMDs and requests as well as handle queries related to the structure of the scenario. Such queries

might me: “give me the IMDs that include video1”, “give me the IMDs that present video1 simultaneously with image1”.

- “*Import” other document formats*. Extent the parser module (see Figure 1.) so that documents resulting from popular authoring tools or other Multimedia Document Standards (SMIL, MHEG) may be stored in our system. This procedure would involve development of translators of such documents to the IMD model that serves as the basis of our system. This process could be eased by swapping the proprietary format currently used with one based on XML, which would also allow the system to take advantage of a number of Java-XML parsers.

The applicability of the approach presented in this paper is wide. The application domains include all the areas where interactive synchronized multimedia content is desirable. As it is apparent the requirements for such content is increasing along with the usage of WWW. As for the rendering mechanism, being platform independent and covering crucial issues, such as interaction, spatial and temporal synchronization, wide distribution features, makes it suitable for generic WWW enabled usage.

REFERENCES

[Buf96] J. Buford, “Evaluating HyTime: An Examination and Implementation Experience”, Proceedings of ACM Hypertext ’96 Conference, 1996.

[Can96] K. Candan, B. Prabhakaran, V. Subrahmanian, “CHIMP: A Framework for Supporting Distributed Multimedia Document Authoring and Presentation”, Proceedings of the fourth ACM international multimedia conference, Boston Ma, 1996, pp. 329-340.

[Hua97] C.-M. Huang and C. Wang, “Interactive Multimedia Communications at the Presentation Layer”, in the proceedings of IDMS’97 workshop, Darmstadt, Germany, 1997, LNCS 1309, pp. 410-419

[ISO93] ISO/IEC, “Information Technology - Coded representation of Multimedia and Hyper-media Information Objects (MHEG)”, 1993.

[Jav97-1] Java – Remote Method Invocation, available at: http://java.sun.com/products/jdk/rmi/

[Jav97-2] Java – Media Framework available at: http://www.javasoft.com/products/java-media/jmf/

[Jon97] T. Johnson, A. Zhang, “A Framework for Supporting Quality-Based Presentation of Continuous Multimedia Streams”, Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS’97), Ottawa, Canada, June 1997, pp. 169-176.

[Kar96] A. Karmouch, J. Emery, “A playback Schedule Model for Multimedia Documents”, IEEE Multimedia, v3(1), pp. 50-63, 1996.

[Ma96] W. Ma, Y. Lee, D. Du, M. McCahill, “Video-based Hypermedia for Education-On-Demand”, Proceedings of the fourth ACM international multimedia conference, Boston, 1996, pp. 449-450.

[Man97] D. Manolescu, K. Nahrstedt, “Link Management Framework for Hypermedia Documents”, Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS’97), Ottawa, Canada, June 1997, pp. 549-556.

[Nan97] J. Nang, S. Kang, “A New Multimedia Synchronization Specification Method for Temporal and Spatial Events”, Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS’97), Ottawa, Canada, June 1997, pp. 236-243.

[Pat96] K. Patel, D. Simpson, D. Wu, L. Rowe, “Synchronized Continuous Media Playback Through the World Wide Web”, Proceedings of the fourth ACM international multimedia conference, Boston Ma, 1996, pp. 451-452.

[Smi96] B. Smith, L. Rowe J. Konstan, K. Patel, “The Berkeley Continuous Media Toolkit”, Proceedings of the fourth ACM international multimedia conference, Boston Ma, 1996, pp. 451-452.

[SMI98] Synchronized Multimedia Integration Language (SMIL) 1.0 Specification. W3C, Proposed Recommendation.http://www.w3.org/TR/REC-smil/

[Sch97] H. Schulzrinne, A. Rao, R. Lanphier, “Real Time Streaming Protocol (RTSP)”, http://www.realaudio.com/prognet/rt/protocol.txt, 1997.

[Sta97] I. Stamati, M. Trafalis, M. Vazirgiannis, M. Hatzopoulos, “Event Detection and Evaluation in Interactive Multimedia Scenarios - Modeling And Implementation”, Technical Report, Dept of Informatics, University of Athens, Greece, 1997

[VB97] M. Vazirgiannis, S. Boll, "Events In Interactive Multimedia Applications: Modeling And Implementation Design", in the proceedings of the IEEE - ICMCS’97, June 1997, Ottawa, Canada

[VTS98] M. Vazirgiannis, Y. Theodoridis, T. Sellis, "Spatio-Temporal Composition and Indexing for Large Multimedia Applications", in ACM/Springer-Verlag Multimedia Systems Journal, vol. 6(4), 1998

[VTM+00] M. Vazirgiannis, D. Tsirikos, Th. Markousis, M. Trafalis, Y. Stamati, M. Hatzopoulos, T. Sellis, "Interactive Multimedia Documents: a Modeling, Authoring and Rendering approach", to appear in Multimedia Tools & Applications Journal (Kluwer Academic Publishers), 2000