A Client-Server Design for Interactive Multimedia Documents Based on Java

D. Tsirikos[1], T. Markousis[1], Y. Mouroulis[1], M. Hatzopoulos[1], M. Vazirgiannis[2], and Y. Stavrakas[3]

[1] Dept. of Informatics, University of Athens, HELLAS

[2] Dept. of Informatics, Athens Economic & Business University, HELLAS

[3] Dept of El. & Comp. Engineering, National Technical University of Athens, HELLAS

Abstract. In this paper we present the design and implementation of a client-server system for Interactive Multimedia Documents (IMDs). IMDs are based on a well-founded theoretical model that covers the issues of interaction and spatiotemporal synchronization of multimedia objects that are presented according to an IMD scenario. The scenaria reside on a server, while the media objects are distributed over the Internet. The client retrieves a scenario from the server, requests the appropriate media from the corresponding http servers and subsequently resents the scenario according to the specifications it defines. In this respect the client uses events (simple and complex) to track the state of each media object and manipulate it. The whole framework has been implemented in Java using the RMI (Remote Method Invocation) client server communication protocol and the JMF (Java Media Framework) for handling multimedia objects. The system presents a promising approach for distributed interactive multimedia on the Internet and intranets.

Keywords. Interactive Distributed Multimedia Applications, Interactive Scenario Rendering, Multimedia Synchronization, Java.

1. Introduction

Multimedia technology has greatly affected the way we interact with computers. But although multimedia capabilities have become something we expect—and get—from most software applications, the exponential growth of the Internet is bearing a second wave of change, in the form 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. The issue of distributed Interactive Multimedia Documents (IMDs) and, specifically, their retrieval and execution, is an issue of current research [9,4]. However, most of the proposed systems suffer from limited interaction support, both at the modelling and at the implementation level.

In order to create and present an IMD we have defined a strong theoretical model. An IMD is defined in terms of *actors*, *events* and *scenario tuples*. The actors represent the participating media objects and their spatiotemporal transformations. An actor does not contain the media to be presented; instead it 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, actor state changes or by the system. In [18] the reader may find details about a rich model for events in IMDs. The primary constituent of an IMD is the scenario (i.e. a set of scenario tuples). A tuple is an

autonomous entity of functionality in the scenario, conveying information about which event(s) start (or stop) an instruction stream. The latter is a set of synchronised media presentations, i.e. an expression that involves Temporal Access Control (TAC) actions on actors, such as *start, stop, pause,* and *resume*. In [19] a set of operators has been defined for the TAC actions and for the corresponding events.

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

2. The Implementation Framework

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. The advantages that arise by that choice are:

- *Platform independence:* The server is written in 100% Java and will run on any environment that supports Java. The client is also written in Java, but makes extensive use of the JMF API. This API uses native (platform dependent) methods and it is currently available for Windows/Intel, Solaris/Sun, IRIX/SGI and MacOS/Macintosh machines. We tested our implementation with Intel's API for Windows, which was the first to be made available.
- *Internet use:* Most current browsers support Java. Shortly the client will run as an applet, it can be downloaded by the browser and used to present scenaria on a separate window. This eliminates the need to install and customise the client, and enables its automatic upgrade without user intervention.
- *Built-in multi-thread support*: With Java it is easy to create multi-threaded programs. The theoretical model we exploited is oriented to concurrent media presentations that interact with each other. Thus implementing each object as a thread, comes quite naturally. Moreover, using separate threads improves robustness since if one of them crashes the rest of the application will continue. This improves the fault tolerance, a necessary measure considering the number of potential pitfalls in a network application.

The communication between the client and the server is performed exclusively using the RMI (Remote Method Invocation) protocol [6], 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 unlike CORBA and DCOM.
- 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 scenaria without any need for complex operations on objects, relations

between them or transaction processing. Furthermore RMI does not require the use of extra software as ORBs and is more efficient than CORBA.

When it comes to presenting media objects we make a distinction between continuous (video and sound) and static (all other kinds of) media. The former are presented using the JMF [7], which specifies a unified architecture, messaging protocol and programming interface for playing and capturing media. Streaming media is supported, where video and sound are reproduced while being downloaded without being stored locally, a feature we exploit extensively. For static media we use the standard Java classes as defined in the Java language without any need for the JMF.

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 (such as [5,19]). 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 scenaria 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 scenaria that range from a few buttons to a full-fledged presentation with lots of sound and images and about 20MB of video. All scenaria were tested over Ethernet LAN's and over a WAN. Areas not yet covered are spatial synchronization relationships and QoS issues.

3. The Server and the Communication Architecture

The servers' task is to provide the client with the scenaria it requests, together with all the media they use. In order to minimise download latency and space requirements at the client side, 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 server system carries out two tasks:

- *The storage of scenaria.* This is handled by the IMD server which is responsible for publishing a list of the scenaria it can provide. At this level, all communication with the client is performed by remote method calls, through the Java RMI registry. All scenaria are implemented as serializable Java objects, stored in a file at the server side.
- *The distribution of media*, which is handled by a set of http servers. The client downloads a scenario, parses and executes it. When some media should be displayed, the client communicates directly with the corresponding http server and retrieves it. In case of static media, these are *first* downloaded and *then* presented according to the scenario specifications. However, when continuous media are to be shown, they are played directly from their site of origin without

being locally stored, using the JMF. The decisive factor on the selection of JMF was that it needs to be present only at the client side; while on the server side, any http server can be used without any special configuration. This allows the massive distribution and replication of media to any number of http servers, thereby reducing the load on any individual server; whereas the scenaria (usually only a few kilobytes in size), can all be served by one machine. Another interesting option, is the ability to use http servers owned and administered outside the scenario creation team.

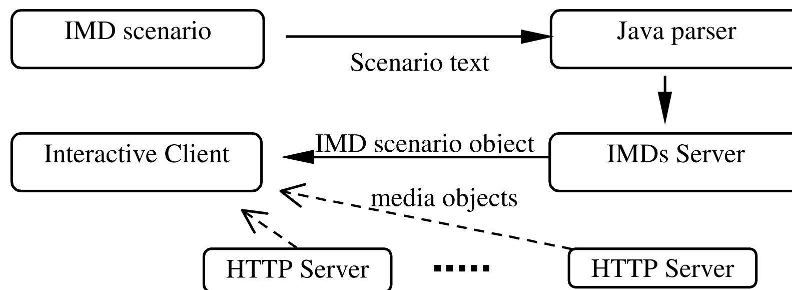

Figure 1. An IMD lifecycle

4. The Client

The client retrieves scenaria from the server, media objects from the appropriate http servers, and presents them according to scenario and interaction. An IMD scenario execution scheme must primarily be able to detect and evaluate events generated by the system, the user or the actors. Other important features are asynchronous activation of tuples, concurrent execution of all instruction streams and synchronised presentation actions according to the scenario tuples' specifications.

Each element of a scenario (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. Each actor is an instance of the *Actor* class that serves as the super-class of *Video*, *Image*, *Sound*, *Text*, *Button*, *Label* and *Timer*. The class *InstructionStreamPlayer* is responsible for synchronised presentation of media objects, while the listeners are in charge of the presentation of a single media object and also of detecting all events related to the object presented. 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 then send messages to the *ScenarioPlayer* indicating which tuples should start or stop.

The outline of the IMD client architecture is shown in Figure 2. When an IMD session starts, a *ScenarioPlayer* and an *EventEvaluator* object are created and the "StartApp" (start application) event is generated. This event is sent to the *EventEvaluator* , which determines which tuple(s) are to be started. *ScenarioPlayer* then creates the corresponding *TuplePlayer* objects that create as many *InstructionStreamPlayer* objects as necessary (remember, a tuple is a set of

instruction streams). Each *InstructionStreamPlayer* contains some actors, each actor holding the presentation specifications for the media object it points to. For each actor an appropriate listener is created. Each object created corresponds to a new thread. During scenario execution, all events generated that are of some interest to the IMD are sent to the *EventEvaluator*, which evaluates them and notifies the *ScenarioPlayer* for changes in tuples' states. Then the *ScenarioPlayer* starts/stops the appropriate tuples either by creating new *TuplePlayer* objects or by destroying the ones that should stop. When a tuple must be interrupted, all the participating actors are interrupted (if they are active) and the corresponding event is sent to the *EventEvaluator*. The client performs two main tasks: starting and interrupting scenario tuples on the basis of occurring events and presenting media objects according to the specifications of the actors. In the following subsections we present these tasks in detail.

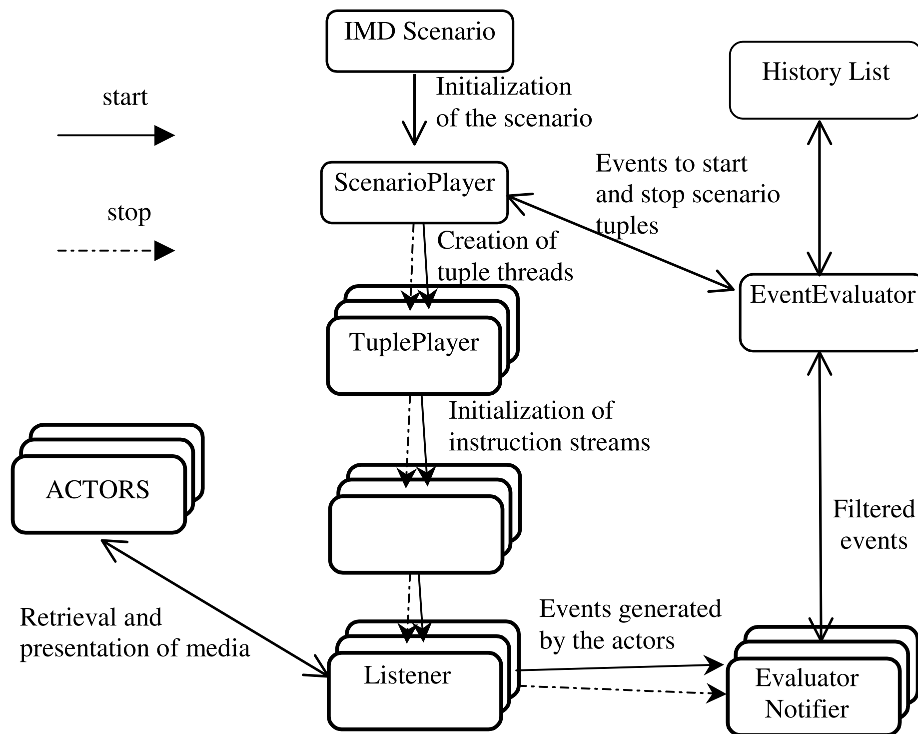

Figure 2. The architecture of the client for IMD scenario execution

4.1 Starting and Interrupting Scenario Tuples

Before proceeding we must distinguish between simple and complex events. Simple events refer to an actor changing state, to events caused by the system or to events originating from the user. Complex events are combinations of simple events, using a

set of operators (e.g. “event e1 AND (event e2 before event e3)”). The only complex events in our design are the start/stop events of the tuples.

For the client application to present a scenario it must detect and evaluate the events that occur in an IMD session and match them against the events included in the “start event” / “stop event” attributes of the scenario tuples. A tuple is considered *active* when the start event of that tuple is evaluated as *true*. At this point all instruction streams of the tuple start executing. Once a tuple has been initiated, it may end in *natural* or *forced* way. In the first 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 stopped. In the second case, the tuple stops when its stop event becomes true. In order to avoid confusion, we explain hereby what are the semantics of interrupting an actor. For this purpose we distinguish between actors with inherent temporal features (sound or video) and actors without such features. An actor of the first category falls in the idle state either when its natural end comes (there is no more data to be presented) or when it is stopped using the stop operator “!” [19]. Actors of the second category (e.g. an image) stop only when we apply the stop operator to them. Hereafter we will examine the classes that are related to management of scenario tuples, namely the *ScenarioPlayer*, the *TuplePlayer*, the *EventEvaluator*, and the *HistoryList* classes.

The *ScenarioPlayer* class is responsible for the execution of an IMD. Initially it constructs the window where the media are to be presented and receives all input events (keyboard or mouse generated) as well as all application timer events. This class is also responsible for starting and interrupting tuples. The *TuplePlayer* class is in charge of starting and interrupting a scenario tuple. In other words, it starts the instruction streams of the scenario tuple with no further effect on them. The *TuplePlayer* must detect the termination of the instruction streams that it contains. When all instruction streams have finished, the *TuplePlayer* informs the *ScenarioPlayer* and then it stops. The *EventEvaluator* class evaluates simple events, and the start and stop events of all the scenario tuples. This means that on arrival of a new event, the start events of all idle tuples and the stop events of all active tuples are evaluated, and those that are found to be true trigger the appropriate action(s). 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. The *HistoryList* class is contained as an attribute in the *EventEvaluator*. There is only one instance of this class in each IMD session; it keeps information on the events that have occurred in an IMD session from the start to the current time, which is defined as the time (in seconds) elapsed since the start of the session. For each event we keep all the timestamps of its occurrences. It is important to clarify that in the *HistoryList* only simple events are stored.

4.2 Presenting the Media

In this section we present the classes of the client that are in charge of presenting the media objects according to the synchronization relationships that are included in the instruction streams. As mentioned above, each scenario tuple consists of a set of instruction streams. Since not all instruction streams have the same effect on the actor

states, we distinguish two categories of instruction streams. The first one includes instruction streams that begin with an actor followed by the start operator (>) and remain active until *all* participating actors stop. The second category includes instruction streams that contain the synchronization operator “/\” and remain active until the temporally shorter of the involved actors finishes executing. If an instruction stream contains the synchronization operator “/\”, it cannot contain any other operator (i.e. >, !, ||, \>). The role of an instruction stream is to translate the synchronization relationships between actors into actual operations on them.

A listener is responsible for presenting a single media object. For that purpose a set of six classes (each for a different kind of actor) were created and all have the suffix “Listener”. These classes do not only present actors, but also detect (“listen to”) any events concerning 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; it can also detect all kinds of events that are related to the particular video. A *VideoListener* must receive appropriate messages from the *InstructionStreamPlayer* to start, pause, resume and stop the video it is currently playing. The class is also in charge of presenting the video according to the specifications in the corresponding actor. The same applies to the other listeners.

Each listener occurrence is paired with an occurrence 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. the ones that the author of the scenario has already defined and are stored in an auxiliary list). The *EvaluatorNotifier* receives messages denoting actor state change, checks whether there is a related event defined by the scenario author 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 is defined by the author. If such an event exists, it will be sent to the *EventEvaluator* together with its 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.

4.3 Event Evaluation

As already mentioned, the interaction in our system is handled in terms of simple and complex events occurring in the IMD context, generated by the user 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 that start/stop each tuple or denote that the IMD session must stop. The evaluation process is carried out as follows. At first, we are interested in the evaluation of a simple occurring event. This task is accomplished by the function `evaluate_simple_event (simple event e)`, that does not actually receive the event, but is called after the event has been received. The method 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 t.stop_event is true
            then add t in tuples_to_stop array
end for
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 goes one more time into the locked state 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 each tuple. The resulting value (true or false) will determine whether tuple `t` should start/stop. When the start/stop event is a simple event, the evaluation is limited to searching in the *HistoryList* for an occurrence of such an event. In case the start/stop event is complex, it may contain expressions including operators and functions that are defined in the framework presented in [19]. Our system implements the following subset of operators: AND, OR, NOT, and of functions: ANY, ANYNEW, IN, TIMES, SEQ and (event1: time_indication : event2).

For complex events the evaluation process is carried out in three distinct steps. The first step is the transformation of the event expression into postfix form resulting in an expression without brackets. The second step is the evaluation of each function appearing in the expression, and replacing the function with the token "true" or "false" according to the result of each function. The last step is to evaluate the result that now consists of the tokens "true" or "false" combined with the Boolean operators.

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 Java's support for threads. Thus, each instance of the classes *Listener, EvaluatorNotifier, InstructionStreamPlayer, TuplePlayer* are served by the corresponding Java thread.

5. Related Work

Regarding Multimedia Document Standards, there have been mainly two efforts: HyTime and MHEG. HyTime [2] provides a rich specification of a spatiotemporal scheme with the FCS (Finite Coordinate Space). HyTime is not an efficient solution for IMD development since "there are significant representational limitations with regard to interactive behaviour, support for scripting language integration, and presentation aspects" [2]. MHEG [5] allows for user interaction between the selection of one or more choices out of some user-defined alternatives. The actual selection by a user determines how a presentation continues. Another interaction type is the modification interaction, which is used to process user input. Object Composition Petri Nets (OCPN) [10] do not allow modelling of interaction.

As mentioned in [1], event based representation of a multimedia scenario is one of the four categories for modelling a multimedia presentation. There it is mentioned that events are modelled in HyTime and HyperODA. Events in HyTime are defined as presentations of media objects along with its playout specifications and its FCS coordinates. All these approaches suffer from poor semantics conveyed by the events and, moreover, they do not provide any scheme for event composition and detection. There has been substantial research in the field of multimedia storage servers, especially video servers. For real-time applications, there is a strong need for streaming media. 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 [15] 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.
- Microsoft Netshow. It is a product of Microsoft 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. NetShow is related to ActiveMovie technology, which is used by the Intel implementation of the JMF.
- Berkeley Continuous Media Toolkit (CMT) [16]. It is a framework that consists of a suite of customisable applications that handle streaming multimedia data. It requires significant programming to support each medium. On this toolkit, cmplayer [14] was built, an application that is used to present remote continuous media in conjunction with a web browser. It supports the synchronised reproduction of multiple media and is available for most platforms.

All the above projects concentrate on how media are transferred and displayed. On the other hand, 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 [8], where algorithms are described for the effective synchronization between media and the maintenance of a QoS.
- 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" [3].
- The system proposed in [12] involves more user participation, since it is 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.
- A similar architecture is presented in [11], 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 [13], 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 scenaria it produces can also be reproduced in different platforms.

To summarise, 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 and is still an open issue.

6. Conclusions

In this paper we have presented a Java-based client-server system for IMDs, supporting a high level of interactivity and distribution of scenario and media. The architecture of the implemented system consists of the IMD scenario server, the media servers and the client module. The IMD scenario server provides the scenaria, while the media objects are distributed in any http server. The client retrieves a scenario from the server and requests the appropriate media from the corresponding http servers. The client design covers widely the issue of interaction with external and internal entities in terms of simple and complex events. Moreover, it maintains the high-level spatial and temporal synchronization requirements of each scenario. The system has been implemented in Java using the RMI client server communication protocol and the JMF for handling multimedia objects.

The salient features of the system presented are:

- *Support for highly interactive presentations,* due to the model exploited in [18, 19]. This model covers internal and external interaction in terms of events. Complex interaction may be covered by composite events using the appropriate composition operators.

- *Platform independence*: platform independent design of a client server system for IMDs. The physical separation of IMD structure (scenario) and content (media), i.e. media to be presented may reside at any http server, allows the usage of external resources for storage and presentation on the IMD content, and reduces the workload and maintenance for the server. The choice of Java (along with accompanying technologies like RMI and JMF) as the implementation platform and the storage of media objects in http servers, makes the design appealing for wide Internet usage. The clients are capable of presenting multiple scenaria simultaneously. The ability to view IMDs using Java applets makes the application available to anyone with a WWW browser.
- *Generic multi-threaded approach for rendering interactive scenaria*. Based on a rich IMD model [18, 19], 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.

Due to the feasibility of running the client module in any WWW browser, the system presents a promising approach for distributed interactive multimedia on the Internet and intranets.

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

- *Provision of QoS*. Provisions could be made to ensure the QoS.
- *Database support at the server 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 handling queries related to the structure of the scenario. Such queries might me: "give me the IMDs that include video1" or "give me the IMDs which include the word "vacation" in at least one of the texts that they present".
- *"Import" other document formats*. Extend the parser module so that documents resulting from popular authoring tools or other Multimedia Document Standards (Hytime, 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.

References

1. Blakowski, G., Steinmetz, R., "A Media Synchronization Survey: Reference Model, Specification, and Case Studies", IEEE Journal on Selected Areas in Communications, vol 14, No. 1, (Jan. 1996), 5-35
2. Buford, J., "Evaluating HyTime: An Examination and Implementation Experience", Proceedings of the ACM Hypertext '96 Conference, (1996)
3. Candan, K., Prabhakaran, B., Subrahmanian, V., "CHIMP: A Framework for Supporting Distributed Multimedia Document Authoring and Presentation", Proceedings of the fourth ACM international multimedia conference, Boston, (1996), 329-340

4. Huang, C.-M., Wang, C., “Interactive Multimedia Communications at the Presentation Layer”, in the proceedings of IMDS’97 workshop, Darmstadt, Germany, (1997), LNCS 1309, 410-419

5. ISO/IEC, Information Technology - Coded representation of Multimedia and Hypermedia Information Objects (MHEG), (1993)

6. Java-Remote Method Invocation, available at: http://java.sun.com:81/marketing/collateral/rmi_ds.html

7. Java–Media Framework, available at: http://www.javasoft.com/products/java-media/jmf/

8. Johnson, T., Zhang, A., “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), 169-176

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

10. Little, T., Ghafoor, A., “Interval-Based Conceptual Models for Time-Dependent Multimedia Data”, IEEE Transactions on Data and Knowledge Engineering, Vol. 5, No. 4, (August 1993), 551-563

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

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

13. Nang, J., Kang, S., “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), 236-243

14. Patel, K., Simpson, D., Wu, D., Rowe, L., “Synchronized Continuous Media Playback Through the World Wide Web”, Proceedings of the fourth ACM international multimedia conference, Boston, (1996)

15. Schulzrinne, H., Rao, A., Lanphier, R., “Real Time Streaming Protocol (RTSP)”, ftp://ftp.isi.edu/in-notes/rfc2326.txt, (1997)

16. Smith, B., Rowe, L., Konstan, J., Patel, K., “The Berkeley Continuous Media Toolkit”, Proceedings of the fourth ACM international multimedia conference, Boston, (1996), 451-452

17. Stamati, I., Trafalis, M., Vazirgiannis, M., Hatzopoulos, M., “Event Detection and Evaluation in Interactive Multimedia Scenaria - Modeling And Implementation”, Technical Report, Dept of Informatics, University of Athens, Hellas, (1997)

18. Vazirgiannis, M., Boll, S., “Events In Interactive Multimedia Applications: Modeling And Implementation Design”, in the proceedings of the IEEE - ICMCS’97, (June 1997), Ottawa, Canada

19. Vazirgiannis, M., Theodoridis, Y., Sellis, T., “Spatio-Temporal Composition and Indexing for Large Multimedia Applications”, to appear in ACM/Springer Verlag Multimedia Systems Journal, September 1998.