Designing a Distributed Multimedia Synchronization Scheduler

Jerzy P. Jarmasz and Nicolas D. Georganas
Multimedia Communications Research Laboratory
Department of Electrical and Computer Engineering
University of Ottawa, Canada.
jerzy@mcrlab.uottawa.ca, georgana@mcrlab.uottawa.ca

Abstract

Distributed multimedia applications need an effective synchronization strategy to deal with the temporal disturbances introduced into the data streams by the transport and operating systems. An essential part of this strategy is scheduling - i.e. determining optimal times for the retrieval of data from the data servers. The CITR News-On-Demand project prototype currently uses a centralized synchronization system which lacks effective high-level scheduling. This paper looks at how to add this functionality without seriously disturbing the components already in place. The proposed solution is a distributed synchronization scheduler, designed and implemented in a way which makes use of the current prototype's components.

1. Introduction

Multimedia technology - still one of the primary areas of research in the computing world - is the ability to integrate the traditional media formats, i.e. text, graphics, sound and moving images, into a single document. The media which were first used in computer applications - text and graphics - are relatively simple in that they have no implicit time semantics or constraints; only their spatial semantics (e.g., where a paragraph should be placed on a screen, or how big the font should be) are relevant to the user. Audio and video, however, are so-called time-continuous or time-dependent media: they only make sense if they are played back in the right order and at the right speed. Displaying and integrating these media types, therefore, requires some kind of temporal coordination. This means that a multimedia application cannot be designed without taking the temporal relationships between its components into account. Multimedia synchronization usually refers to these temporal relationships among the different media [1].

Synchronization errors - disturbances in the temporal semantics of data, such as a varying frame rate for video - are due to all the mechanisms wich get the data from the storage medium to the display devices at the user's end of the application. Briefly, these mechanisms are: the retrieval of data from the storage medium, the transport service between the storage medium and the displaying application, the displaying application itself, and the underlying operating systems and protocols which run the other mechanisms. Synchronization methods can be either preventive or corrective. Preventive mechanisms are designed to minimize synchronization errors as data move through the system to the user. These include scheduling, which is the process of organizing and coordinating the retrieval of data from the storage medium, and efficient transport mechanisms and operating systems, which allow for the precise control of timing constraints. Corrective methods involve adjusting the presentation of the data once they are received by the displaying application so that the appearance of synchrony is maintained. Synchronization strategies in multimedia applications use these methods in varying degrees.

An example of such an application is the News-on-Demand system being developed as part of the research in the Broadband Services major project of the Canadian Institute for Telecommunications Research (CITR). This project is made up of many sub-projects being carried out by different research teams. The sub-projects include a file system for time-dependent media (the Continuous Media File Server, or CMFS), a Quality-of-Service negotiation and monitoring system, database technology and synchronization of data.

The synchronization research is being done by a team from the Multimedia Communications Research Laboratory (MCRLab), of the University of Ottawa's Electrical and Computer Engineering department. The MCRLab's synchronization strategy looks at both the preventive and corrective aspects of synchronization. Prevention is mainly at the scheduling level. Due to technical considerations, the current overall project prototype uses a centralized design for the scheduler [2].

This means that only one module, located at the client, organizes and coordinates the retrieval of the data form the storage medium - in this case, the CMFS and the text database. However, there is research [3] which suggests that a distributed scheduler, where the servers participate in the scheduling of the multimedia data, would be more efficient, particularly for the client. The MCRLab has therefore committed itself to re-designing the scheduler in the current prototype to make it distributed.

This paper will therefore study the issues involved in changing from a centralized to a distributed scheduler and the implementation of such a scheduler. The rest of the paper is organized as follows. Section two studies synchronization issues. Section three presents and compares different scheduler architectures. Section four discusses what needs to be changed in the current prototype to move to a distributed scheduler. Section five discusses the actual implementation of the scheduler. Section six points to further research and concludes the paper.

2. Synchronization Concepts

2.1 Media granularity and semantics

There can be many different types of temporal relations between the components of multimedia data. A good way to classify them is to look at the different semantic levels of multimedia data. A reference model for organizing these levels has been given in [1]. Briefly stated, this model recognizes three levels of organization inside multimedia documents, and one level "above" the document, which allows the author of the document to specify temporal relations. The levels inside a document are:

- the media layer, which looks at the semantics inside a single stream of time-dependent data. The stream is viewed as a succession of Logical Data Units, or LDUs, which denote the lowest level of data granularity seen by the application. This could be frames, groups of frames, data blocks, etc. Synchronization at this level is referred to as intra-stream synchronization.
- the stream layer, which looks at the relations between whole streams, and groups of streams. LDUs are hidden at this level, streams are seen as a whole. Synchronization at this level is known as inter-stream synchronization.
- the object layer, which integrates streams and time-independent data such as text and still images. The difference between the two types of objects are hidden here. Synchronization at this level is called inter-object synchronization.

These levels allow us to organize and study the types of temporal constrains between multimedia data, and the synchronization errors that occur during the playback of a multimedia document.

2.2 System components and how they affect synchronization.

As it was said before, there are many links in the chain which moves data from its storage site to the user. Each of these links performs a certain task and affects the data in a different way. They all tend to introduce synchronization errors, unless care is taken to avoid this. Servers store data and put that data into the network; the network transports the data to the client; the client reads the data from the network and presents it to the user; operating systems and protocols allow these systems to run and do their work.

Obviously, given the special temporal constraints of multimedia data, as well as their resource requirements (high bandwidth and high resolution for good video), the systems described above have to be designed or chosen with care. In particular, the servers must be able to store large amounts of data in such a way that retrieval is quick and efficient; networks must provide high bandwidth and low transmission rate variations; operating systems and applications must be able to provide real-time data processing (retrieval, re-synchronization, display) [4].

Synchronization methods usually aim at making the best of the properties and functionalities of the "infrastructure" an application is using, not to change them; the synchronization research at the MCRLab thus concentrates mainly on server- and client- side application-level strategies.

2.3 Synchronization strategies

2.3.1 Prevention. Preventing synchronization errors involves minimizing these errors. This requires eliminating any inefficiencies and latencies in the mechanisms which move data from the storage medium to the user. These mainly involve disk-reading scheduling policies, network transport protocols and operating systems which allow for real-time computing and for temporal semantics of data to be taken into account.

Looking at the data semantics introduced earlier, it can be said that "infrastructure" systems take care of intra-stream synchronization, while applications usually use this "infrastructure" to ensure inter-stream and inter-object synchronization, although some operating systems might provide facilities for inter-stream synchronization [1] and some applications might delve into intra-stream synchronization. At any rate, the prevention methods developed at the MCRLab concentrate on providing inter-object scheduling using lower-level mechanisms and a temporal notation called Time Flow Graphs (TFG) [5]. A scheduler uses the specifications for a presentation described with this notation to create a schedule for the delivery of the objects to the client by the servers (delivery schedule) and the presentation of these objects to the user by the client application (presentation schedule) [6]. This scheduler can be centralized - entirely located at the client - or distributed - the delivery scheduling functionalities are shared among the servers and the client. So far research into the distributed scheduler has been mainly theoretical; later sections of this paper will discuss the actual implementation of such a scheduler.

2.3.2 Correction. If errors can't be prevented - and this being the real world, they can't - they must be rectified. This is to be done by the client application receiving the data. The method used by the MCRLab is the Stream Synchronization Protocol (SSP)[2]: units which control and monitor the client-end of the data connections compare the real arrival times of data with the ones predicted by the presentation schedule and notify the scheduler of any discrepancies. These discrepancies are then compensated for by the scheduler, which delays the display of data that are "ahead" of other data, allowing the late data to "catch up." Since this is done entirely at the client side, the architecture of the scheduler does not affect the SSP, and the SSP does not affect the servers.

3. Scheduler Models

3.1 The centralized scheduler

As has been previously stated, in this model all of the object-level scheduling is performed at the client by a process called a scheduler. When a user requests an article, the scheduler receives from the database server the scenario for the article. The scheduler then computes a presentation schedule which fits the scenario, as well as a delivery schedule for the servers, which is basically the presentation scenario adjusted for network conditions. The client decides when it wants to present data and when it wants to request the data from the servers. The servers take care of low-level (i.e. intra-stream) scheduling only. The client then takes care of high-level and low-level synchronization with the SSP. Figure 1 shows a diagram of the current system prototype that has been implemented using a centralized scheduler [7].

The user interacts with the application through a Graphical User Interface (GUI). The user first chooses an article from the list of available articles that is presented to him/her by the GUI. This request is then sent to the Database Server. The Database Server returns the article scenario to the scheduler, which takes care of all scheduling tasks (starting data readers, computing schedules, and so on) in the client application. The scheduler uses the scenario to compute an acceptable presentation schedule. Following this schedule, the client

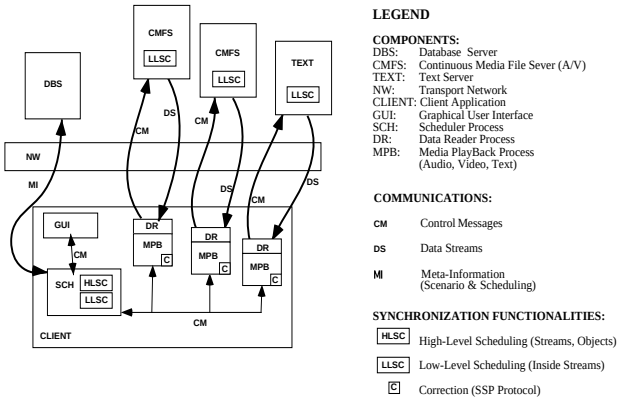

Figure 1: System prototype with a centralized scheduler

opens connections between itself and the servers and requests the different media that needs to present the article. There is no explicit delivery schedule in this model. When the user gives the command to start displaying the article, the client starts reading data from the connections and displaying them. The data streams are constantly monitored by the data readers, which have a Media Synchronization Controller (MSC); this MSC knows the presentation schedule and notifies the scheduler when synchronization errors (with regards to the schedule) are detected. The scheduler then re-schedules the reading or the display of certain streams, using the Stream Synchronization Protocol [7].

It is clear from the figure that control over the system is centralized at the client: the client determines the schedules, opens the connections and requests the data. This type of architecture requires the client application to be quite intelligent. Since it is desirable that the client

application be kept simple, so that it will easily run on a wide variety of platforms, this should be avoided. Another drawback is that there is no proper high-level (i.e. stream and object layer) scheduling in this model, only correction measures. Servers with some kind of high-level scheduling capability would help reduce synchronization errors, thus lightening the workload of the client. As it stands, the client does all of the synchronization work, except for media layer delivery scheduling.

3.2 The distributed scheduler

Research conducted at the MCRLab suggests that a distributed scheduling scheme provides better synchronization prevention and recovery performance than a centralized system. A distributed architecture was proposed in [3]; its impact on scheduling was discussed in [8]. In this architecture, the servers become responsible for opening the data connections, obtaining information on network conditions and scheduling the delivery of their own data. The scheduling at the server level is performed by Temporal Scheduler Controllers (TSCs). The presentation scenario is sent by the database server to the client and to the TSCs. The TSCs use the scenario to open the appropriate connections to the client; in doing so, they obtain the end-to-end network delay on each connection. The TSCs then exchange information about the different objects in the presentation and network conditions to derive a presentation schedule and a delivery schedule. The delivery schedule, i.e. the optimal times for transmitting data to the client, is derived from the presentation schedule by subtracting the total delay that each object encounters from the times in the presentation schedule. The total delay for each object is the sum of the end-to-end network delay on its connection, the buffer delay upon reception by the client, and the decoding time for the object [6] [7]. The presentation schedule is also derived at the client from the scenario and the TFG for the presentation.

The connection-endpoints are controlled by Media Synchronization Controllers (MSCs), which ensure intra-stream synchronization. The server MSCs (SMSCs) are directly responsible for opening the connections requested by the TSCs and for ensuring media layer scheduling for each stream. The client MSCs (CMSCs) are in charge of monitoring synchronization errors upon reception of the streams by the client. Errors are detected by comparing the actual arrival time of an object or an LDU with its projected arrival time based on the schedule provided by the client scheduler. The scheduler is notified of any errors and re-schedules the rest of the presentation accordingly. For example, if an object arrives late, the scheduler delays the presentation of objects which are to be presented either in parallel with, or after, the late object. This co-operation between the scheduler and the CMSCs implements the SSP correction protocol alluded to in section 2.3.2 and explained in more detail in [7].

The overall architecture of this system is illustrated in figure 2, with a particular emphasis on control messages between the components .

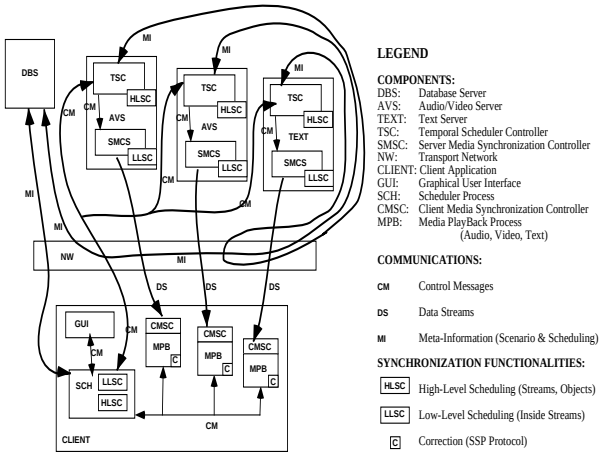

Figure 2: System architecture with a distributed scheduler

Clearly, the servers have a more complicated structure in this model, since they have an extra scheduler module, effectively giving them a two-tiered architecture. Communications between the components are also more complicated, as all the servers must exchange information. But while the servers must now perform more tasks, the client does not have to derive a delivery schedule any more, and the system has stream and object layer scheduling capabilities, which are absent from the centralized scheduler architecture seen in the previous section. This should improve performance by reducing synchronization errors, therefore making recovery operations more efficient, and by simplifying the client application.

4. Re-engineering the Current Model

4.1 Assessing the centralized model

After deciding what the requirements of the new architecture are to be, i.e. a distributed scheduler shared between the client and the servers, the centralized model

has to be assessed to see how similar it is to the distributed model.

A quick visual comparison of figures 1 and 2 reveals that the low-level (i.e. intra-stream) scheduling that is required in the distributed model is provided by the CMFS servers in the centralized model. In other words, the SMSCs of figure 2 are equivalent to the CMFSs of figure 1. The client appears to have all it needs, since it has the same components in both figures. The main difference is that the servers in the distributed model are more complicated: they include TSC modules for high-level (inter-object and inter-stream) scheduling, absent from the centralized model, which control the SMSC/CMFS units, responsible for low-level scheduling as well as network functionalities. So moving from the centralized to the distributed model requires changing the concept of a server from just the CMFS to a server with high-level scheduling capabilities which utilizes the functionalities of the CMFS for low-level synchronization. All that the centralized model is missing is some way of coordinating the CMFS functionalities from a location other than the client application.

4.2 The new distributed model

Figure 3 illustrates the proposed architecture for the implementation of the distributed scheduler. What changes from the centralized model is that TSC modules from the theoretical distributed model which communicate with the CMFSs are added to the servers. These TSCs act as proxies between the client and the servers; the client doesn't communicate directly with the CMFSs anymore. To allow the TSCs to coordinate network connections and data streams through the CMFS, some changes were made to the CMFS by its authors [9], as follows:

- the CMFS API now allows separate processes to set up and control the server and the client end-points of data connections; this allows the TSC to manage the server end-point, while the client application manages the client end-point

- streams do not have to be readied for transmission close to the desired transmission time; rather, they can now be readied in advance, allowing for more flexible inter-stream scheduling.

Both these changes allow the TSC to carry out inter-stream delivery scheduling for the CMFS servers instead of the client.

However, the original theoretical model has been somewhat simplified for ease of implementation. In the original model, the servers were to communicate with

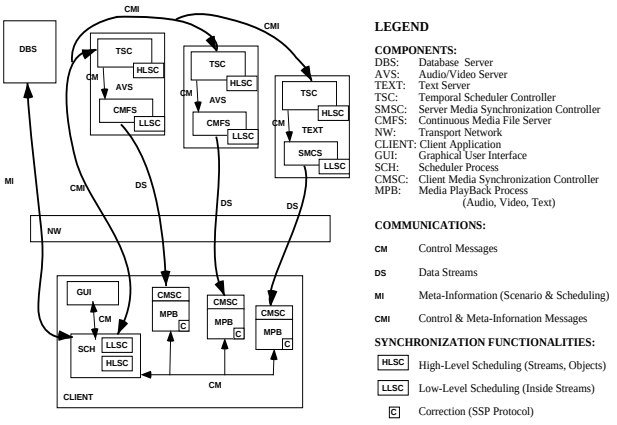

Figure 3: System architecture with proposed distributed scheduler

each other to derive the delivery schedule from the presentation schedule. Indeed, some early attempts at implementation even proposed having the servers derive the TSC and presentation schedules, independently of the client. This would have introduced unnecessary redundancy as well as the problem of coordinating the servers. The approach that was finally chosen requires the servers to receive the presentation schedule from the client and derive the delivery schedule only for the objects that are stored on them. This simplifies the design of the TSCs, and therefore of the servers. The client combines requests to prepare, start or stop a presentation and meta-information (scheduling and synchronization data) in its messages to the TSC processes sitting on the server entities. This allows the servers to share in the scheduling of multimedia data in a simple and effective way without requiring a major reworking of the different components that make up the News-on-Demand project prototype.

5. Implementation

The TSC module and an API to the module have been implemented. The implementation was done using the C programming language and a real-time threads package called RT Threads [10], which also requires C.

The code for the TSC module is quite simple. Its pseudo-code can be summed up as follows:

-the TSC makes itself known to the CMFS and initializes its data structures
-The TSC waits for a message from the client:
  -**TSCOPEN** message:
    -check if the request can be handled

-if so, assign an identifier to the request and remember the scenario sent by the client with the message, and open the necessary connections between the CMFS and the client using the CMFS API

-report the status of the request to the client (refused, successful or failed)

-**TSCPLAY** message:

-check if the request is for an article which has already been opened

-if so, extract the presentation parameters from the message; using the start times for each stream and the overall scenario, which together constitute the presentation schedule, derive the delivery schedule; request that the CMFS start sending the data streams according to the delivery schedule

-report the status of the request to the client

-**TSCSTOP** message:

-check if the request is for an article which is playing

-if so, stop all of the active streams using the CMFS API

-report the status of the request to the client

-**TSCCLOSE** message:

-check if the request is for an article which has been opened

-if so, close all the streams using the CMFS API

-report the status of the request to the client

-The TSC goes back to waiting for a new message

The CMFS consists of two types of processes: the cmfsadm and the cmfsnode. The cmfsnode manages the individual data connections, while the cmfsadm coordinates the cmfsnodes and directs client requests to the appropriate cmfsnode process. There can be many cmfsnodes associated to a cmfsadm process, but a CMFS only contains one cmfsadm. The CMFS API has been designed in such a way that the client is only aware of the cmfsadm process; the CMFS returns the location of the relevant cmfsnodes to the CMFS API, but this information is not forwarded to the client application. To keep the distributed scheduler architecture consistent with this, the TSC makes itself known to the CMFS; the client can thus query the cmfsadm to find out where to find the TSC.

All communications between the client and the TSC are done using specially designed message data structures; these structures include a message type and parameters which are appropriate for the message type. The message types for communications from the client to the server are the ones given in the pseudo-code above. Their associated parameters are as follows: the scenario is sent with TSCOPEN messages; presentation parameters for each stream (start times, playback speed, start and stop positions) and an article identifier are associated to TSCPLAY messages ; an article identifier is sent with the TSCSTOP and TSCCLOSE messages. This article identifier allows the TSC to keep track of the clients which are sending requests to it, as a client presents only one article at a time to the user.

The internal data structures of the TSC are the Document structure, which combines the scenario and the delivery schedule, and which is a simplified version of a similar structure used by the client, and a structure for the presentation parameters of each stream, as well as a table for keeping track of the articles being scheduled by the TSC. The algorithm used to derive the delivery schedule is the one given in section 3.2.

The TSC API allows the client application to communicate with the TSC. The API formats the data given to it by the client for the TSC and sends it using the messages described earlier. In particular, it must take the client's Scenario data structure and send only the information needed by the TSC, which is information at the specification, object and stream layers (see section 2.1). The TSC returns the result of the processing of the

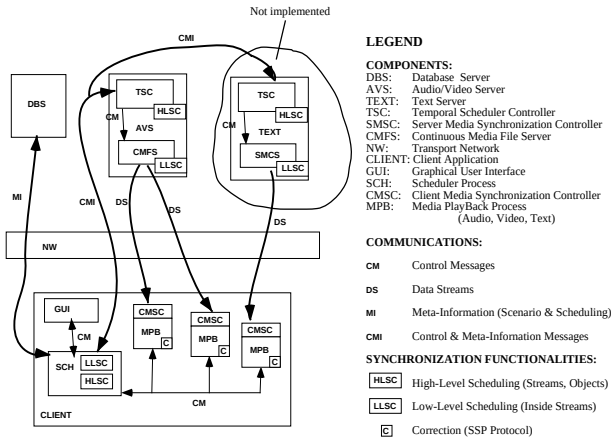

Figure 4: Implemented system with a distributed scheduler

messages to the client using specially formatted reply messages. These contain a status code, and in the case of the processing of a TSCOPEN message, the return message also contains an article identifier and the location of the TSC. The API then transmits the status code to the client application.

The CMFS is also designed so that a client can only use one CMFS server at a time; thus the system only has one time-dependent media server. Also, the only text used in the presentation are text captions synchronized

with the audio, and were thus stored on the CMFS. Figure 4 illustrates the system that was implemented. The distributed scheduler has been integrated into the overall News-On-Demand system. Fine-tuning of the scheduling algorithm, as well as performance evaluation and comparison with the centralized model, can now proceed.

6. Conclusion

A theoretical analysis of the CITR News-on-Demand prototype using a centralized synchronization scheduler has revealed a number of shortcomings. We have seen how these shortcomings can be corrected with a distributed scheduler, as well as how this scheduler can be integrated into the News-On-Demand project without requiring a major re-working of the project. Further research into the performance of the scheduler will reveal if it actually offers a practical advantage over the centralized model. However, informal observations of the performance of the distributed scheduler seem to indicate that the processing load of the client is noticeably reduced with this architecture [11].

This is only a preliminary implementation. Further work on the scheduler should look into fast-forward and rewind functionalities. Research into adapting the scheduler to a multi-server environment is also needed, particularly finding an efficient strategy for coordinating many TSCs in the task of scheduling a presentation whose components are located on many different servers.

Acknowledgments

This research was supported in part by the Canadian Institute for Telecommunications Research (CITR), under the Networks of Centres of Excellence program of the Government of Canada, and in part by a Natural Science and Engineering Research Council (NSERC) postgraduate scholarship

Thanks also go to the CMFS team, especially Dr. Gerald Neufeld, Dr. Norm Hutchinson, Dwight Makaroff and Roland Mechler, from the Distributed Systems Group at the Department of Computer Science at the University of British Columbia for helping us understand their code and for their useful comments and suggestions about the design of the distributed scheduler, as well as for making the first author's stay at their research facilities a pleasant and productive one.

References

[1] Steinmetz, R. and Nahrstedt, K. (1995). *Multimedia: Computing, Communications and Applications*, Chapter 15: Synchronization. Prentice Hall P T R.

[2] Georganas, N.D. (1996). *Synchronization Issues in Multimedia Presentational and Conversational Applications*. In Proceedings of the 1996 Pacific Workshop on Distributed Multimedia Systems (DMS'96), Hong Kong, June 1996 (Invited paper).

[3] Lamont, L., Li, L. and Georganas, N.D. (1994). *Centralized and Distributed Architectures for Multimedia Presentational Applications*. In Broadband Islands '94, Connecting with the End-User, Proceedings of the 3rd International Conference on Broadband Islands, Hamburg, Germany, 7-9 June, 1994, pp. 59-70. Elsevier Science B.V., Amsterdam, The Netherlands.

[4] Burkow, T. M. (1994). *Operating System Support for Distributed Multimedia Applications; A Survey of Current Research.* Technical Report (Pegasus Paper 94-8). Faculty of Computer Science, University of Twente.

[5] Li, L., Karmouch, A. and Georganas, N.D. (1994). *Multimedia Teleorchestra with Independent Sources: Part 1 - Temporal Modeling of Collaborative Multimedia Scenarios.* In ACM Journal of Multimedia Systems, vol. 1, no. 4, February 1994.

[6] Li, L., Karmouch, A. and Georganas, N.D. (1994). *Multimedia Teleorchestra with Independent Sources: Part 2 - Synchronization Algorithms*. In ACM Journal of Multimedia Systems, vol. 1, no. 4, February 1994.

[7] Lamont, L., Li, L., Brimont, R. and Georganas, N.D. (1996). *Synchronization of Multimedia Data for a Multimedia News-on-Demand Application*. In IEEE JSAC , Vol. 14, No.1, Jan. 1996, pp.264-278.

[8] Li, L. (1994). *The Design and Implementation of a Real-time Multimedia Synchronization Control System over High-speed Communications Networks*. M.A.Sc. Thesis. Department of Electrical and Computer Engineering, Faculty of Engineering, University of Ottawa.

[9] Makaroff, D., Hutchinson, N. and Neufeld, G. (1996). *The UBC Distributed Continuous Media File System*. Interface Document, Department of Computer Science, University of Brithish Columbia.

[10] Finkelstein, D., Hutchinson, N.C., Makaroff, D.J., Mechler, R. and Neufeld, G.W. (1995). *Real Time Threads Interface*. Technical Report, Department of Computer Science, University of British Columbia.

[11] Jarmasz, J. (1997). *Notes on Performance Testing and Comparison of the centralized and Distributed Schedulers for the CITR News-on-Demand System*. MCRLab technical report. Multimedia Communications Research Laboratory, Department of Electrical and Computer Engineering, Faculty of Engineering, University of Ottawa.