INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Université d'Ottawa · University of Ottawa

Design and Implementation of a Distributed Synchronization Scheduler for a Multimedia News-on-Demand Application.

Jerzy P. Jarmasz

A thesis submitted to the School of Graduate Studies and Research in partial fulfillment of the requirements for the degree of

MASTER OF APPLIED SCIENCE

Ottawa-Carleton Institute of Electrical Engineering
Department of Electrical and Computer Engineering
Faculty of Engineering
University of Ottawa

February 7, 1997

© Jerzy P. Jarmasz

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced with the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-20924-5

Canadä

// ABSTRACT

The CITR News-On-Demand is a system whose architecture follows the client-server paradigm: a distributed database provides multimedia news articles to users by means of a client application. Due to practical considerations, the synchronization system used in this prototype is a centralized one, in which the client application bears the whole burden of scheduling data retrieval and presentation. While this is simpler to implement, it makes more demands on the client's resources than a distributed system. Furthermore, research into the issue suggests that a distributed system would be more efficient at preventing synchronization errors.

This thesis shows how the synchronization system of the News-on-Demand prototype can be re-engineered to make it distributed. It first considers the theoretical concepts behind synchronization systems and provides a framework for doing so simply and in a manner which can be systematically applied to any system. The News-on-Demand prototype is then studied using this framework, and its shortcomings as regard synchronization are discussed. Research at the MCRLab into distributed synchronization systems is also studied with this framework, and it is shown how this research can be applied to the News-on-Demand system. With this, a distributed synchronization system for the CITR project is designed and implemented, which integrates the distributed synchronization concepts into the existing system. The pseudo-code and the data structures of the new system are given and discussed.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. Georganas, for providing me with a comfortable and well-equipped, if not actually luxurious, environment for my work.

I would also like to thank the CMFS design team at the University of British Columbia, especially Dr. Neufeld, who made sure I had a place to stay and work during my visit to his lab, and Dr. Hutchinson, Dwight Makaroff and Roland Mechler, who answered the questions I had concerning their system and their code and provided welcome tips and suggestions when my own work was stalled by technical problems. I also thank everyone else who made my stay at UBC an enjoyable and profitable one.

Thanks also go to all of my colleagues at the MCRLab, but especially Jeff Brinskelle, who provided much needed technical support and who did not hesitate to explain his work to me when it came time to integrate my part of the project with his.

Of course, the kind helpfulness of Michelle Roy, Andrée Carrière and Lucette Lepage was much appreciated; thanks to them, I was never burdened or concerned with administrative problems.

I am very grateful to NSERC, which ensured my financial security during the course of my studies.

But, most of all, I am forever indebted to my family for their encouragement and their faith in me, and for having provided me with the means to pursue higher studies in the first place.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION 1

1.1 How Data Formats Shape Multimedia Applications 1

1.2 The CITR News-on-Demand Project 3

1.2.1 Project background 3

1.2.2 Original design requirements 4

1.2.3 Current state of the project 5

1.2.4 What needs to be done? 5

1.3 Thesis outline 6

1.4 Publications arising from this research 7

CHAPTER 2: A FRAMEWORK FOR STUDYING THE SYNCHRONIZATION ASPECTS OF MULTIMEDIA APPLICATIONS 8

2.1 Some basic synchronization concepts 8

2.2 Classifying synchronization phenomena - a Synchronization Reference Model 9

2.2.1 The Media Layer 12

2.2.2. The Stream Layer 13

2.2.3 The Object Layer 14

2.2.4 The Specification Layer 15

2.3 Where errors occur - a complete view of the system 16

2.3.1 The Transport System 17

2.3.2 Hardware and Operating Systems 20

2.3.3 The application 22

2.3.4 Review 24

2.4 Synchronization methods: prevention and correction 25
2.4.1 Prevention 25
2.4.2 Correction 26
2.5 Putting it all together - the three axes 27
CHAPTER 3: ANALYSIS OF THE ARCHITECTURE OF THE CURRENT NEWS-ON-DEMAND PROTOTYPE 31
3.1 Overview of the application 31
3.2 Implementation of the architecture 34
3.3 Preventive strategy 39
3.3.1 Server Applications 39
3.3.2 Client Application 40
3.3.3 The Infrastructure 42
3.4 Corrective Strategy 42
3.4.1 The Server Applications 42
3.4.2 The Client Application 43
3.4.3 The Infrastructure 43
3.5 Evaluation of the synchronization capabilities of the current prototype 44
CHAPTER 4: RE-ENGINEERING THE ARCHITECTURE OF THE CENTRALIZED-SCHEDULER PROTOTYPE 47
4.1 The MCRLab's Scheduling Strategy - A Distributed Scheduler 47
4.1.1 Overview of the distributed model 47
4.1.2 The algorithms behind the scheduler 49
4.1.3 Functionalites provided by the distributed scheduler 55
4.2 Moving from a Centralized to a Distributed scheduler 57
4.2.1 Similarities between both models 57
4.2.2 What needs to be changed in the current system 58

CHAPTER 5: IMPLEMENTING THE DISTRIBUTED SCHEDULER 60

5.1 Designing the Architecture 60

5.1.1 Adapting the distributed scheduler to the News-on-Demand system....60

5.1.2 The architecture of the new system.....63

5.2 Changes to the CMFS 66

5.3 New Data Structures Used in the Scheduler 68

5.4 Implementation of the TSC.....69

5.4.1 General requirements.....69

5.4.2 Functioning of the TSC.....70

5.4.3 Delivery schedule algorithm.....72

5.5 Implementation of the TSC API 74

5.5.1 TscOpen 75

5.5.2 TscPlay 76

5.5.3 TscStop and TscClose 77

5.5.4 API implementation 79

5.6 Integration with the Overall Prototype.....79

5.7 Functionalities of the Distributed Scheduler System 81

CHAPTER 6: CONCLUSION 83

6.1 Summary of Thesis 83

6.2 Contributions of this Thesis Work.....84

6.2.1 Implementation of a distributed scheduler.....84

6.2.2 A conceptual framework for studying synchronization systems.....85

6.2.3 Introduction of a new server architecture.....86

6.3 Future Work 87

6.3.1 Performance testing 87

6.3.2 Improving the handling of many clients by the TSC.....88

6.3.3 Implementing TSCs for systems with more than one server 89

6.3.4 The distributed scheduler and user interactions 90

REFERENCES 93

APPENDIX A - DATA STRUCTURES 99

A.1 Scheduling Structures Used by the TSC..99

A.2 New Scheduling Structures Used by the Client.................................101

A.3 Control Messages..102

LIST OF FIGURES

Figure 1: Four Layer Synchronization Reference Model (from [SN95]) 11
Figure 2: Using the three-axis framework to represent a multimedia system 28
Figure 3: Architecture of Prototype with Centralized Scheduler (from [Geo96]) 34
Figure 4: Communications Between the Components of the System 38
Figure 5: Analysis of the Centralized Scheduler System Architecture 45
Figure 6: A Distributed Scheduling Algorithm (from [LLG94]) 48
Figure 7: TFG model of a simple scenario 51
Figure 8: OCPN model of the scenario 52
Figure 9: Analysis of the Distributed Scheduler System Architecture 56
Figure 10: Two-tiered server architecture 61
Figure 11: Implemented architecture of the distributed scheduler 63
Figure 12: State diagram of the TSC 70
Figure 13: Analysis of the Implemented Distributed Scheduler System 82

LIST OF TABLES

Table 1: Correspondence between the components of both systems 59

LIST OF ACRONYMS

API	Application Programming Interface
ATM	Asynchronous Transfer Mode
CITR	Canadian Institute of Telecommunications Research
CMFS	Continuous Media File Server
CMSC	Client Media Synchronization Controller process
GUI	Graphical User Interface
IP	Internet Protocol
LDU	Logical Data Unit
MCRLab	Multimedia Communications Research Laboratory
MSC	Media Synchronization Controller
MT	Media Transport
OCPN	Object Composition Petri Net
QoS	Quality of Service
RPC	Remote Procedure Call
RTT	Real Time Threads
SMSC	Server Media Synchronization Contriller process
SSP	Stream Synchronization Protocol
TCP	Transmission Control Protocol
TFG	Time Flow Graph
TSC	Temporal Scheduler Controller
UDP	User Datagram Protocol

CHAPTER 1: INTRODUCTION

1.1 How Data Formats Shape Multimedia Applications

One of the defining phenomena of the 20th century is what is commonly referred to as the *Information Revolution*. We have always been looking for ways to store, communicate and process information more efficiently. Many devices and techniques for doing so have been imagined and designed over the centuries, but until the necessary technologies were developed, most of these ideas either remained on paper or were simply forgotten. The recent development of computer software and hardware has finally afforded us the technological medium in which to materialize our drive to make the handling of information more efficient.

One of the very recent technologies which has allowed us to rethink the way we access and broadcast information is referred to as "multimedia„. Multimedia computer applications allow for the integration of many types of data, or media, in the same presentation or document. The media are text, graphics, still images, audio and video. Text is the least demanding medium. It is the one that was first used in computing applications and communications. Graphics require somewhat more resources, but basically all that was needed for integrating still images into the world of computers were more performant graphics cards and monitors, more memory and higher network bandwidth. Audio and video, however, present special problems. They are time-dependent media, which means

that they have a temporal as well as a spatial component. Thus audio and video present a special challenge: not only do they require huge computing resources, but they also have specific requirements when it comes to the "lay-out„ of a multimedia document. In other words, the integration of time-dependent data has required the creation of a new field: multimedia.

Multimedia applications are therefore designed with the goal of presenting many different media in an integrated fashion. An example of this is a news service, or a **News-on-Demand** application. Instead of depending on television or radio programs, which may be broadcast at inconvenient times, or on the printed media, which aren't always convenient to read in today's fast-paced workplace, News-on-Demand applications aim at providing users with fast and easy access to the latest news, regardless of medium or format. This is not meant to be merely a fancy, high-tech version of the traditional TV broadcast; news agencies will now have the ability to use the format or formats most appropriate to the information they want to transmit. Multimedia news documents should be able to combine the advantages of all of the traditional media: text and graphics provide a convenient and cost-effective way to communicate a lot of background and in-depth information, as well as large amounts of simple data (facts, numbers, etc.), while audio and video provide emotional content as well as a sense of immediacy, of "being there.„ It is clear from this that a multimedia News-on-Demand service should be a distributed system - i.e. the information is stored at many different locations - with a fair bit of coordination of the media that make up the particular documents.

1.2 The CITR News-on-Demand Project

1.2.1 Project background

The Canadian Institute for Telecommunications Research (CITR), funded by the federal government, is a research agency - or, more accurately, a network of research centres - whose purpose is to investigate emerging telecommunications technologies for their economic viability and industrial potential. The CITR's research is divided into five main projects. One of those is the Broadband Services major project (1993 -), whose goal is "... to research and prototype enabling technologies for distributed multimedia applications.„ This first target application of this project was the News-On-Demand service. The project has six sub-projects, each being carried out by a different research group. These are:

-database technology for distributed multimedia, at the University of Alberta

-distributed continuous media file systems, at the University of British Columbia

-Quality of Service management, negotiation, monitoring and control, at the University of Montreal

-synchronization technology, at the University of Ottawa (MCRLab)

-architecture and integration, at the University of Waterloo

-scaleable video encoding techniques, at INRS Télécommunications, with scaleable image encoding, at the University of Ottawa (VCCLab)

This thesis will concern itself mainly with synchronization issues.

1.2.2 Original design requirements

The original requirements for the project were:

-a distributed database: this means that the information available to users is located on different servers, presumably on servers best suited to the particular medium or media the data are in

-servers adapted to different types of media: the server must be able to optimize storage, retrieval and delivery of its data depending on the requirements of the data. This would be best achieved if each server is used only for a certain type of data.

-optimizing data transport by using an ATM platform and the Quality of Service (QoS) paradigm

-making the client as simple as possible, so it could be implemented on a wide range of platforms

-distributing the data synchronization mechanisms and algorithms, in part to keep the client simple, in part to optimize synchronization in general. This means that the coordination of the synchronization operations was to be shared among all components concerned.

1.2.3 Current state of the project

Much design and implementation work has already been done. Most of the original requirements either are being met or have been met. However, due to what seemed at the time to be important incompatibilities between the synchronization mechanisms and the design of the Continuous Media File Servers (CMFSs), a centralized synchronization architecture, in which the client coordinates all of the synchronization operations, has temporarily been adopted. The prototype of the system as presently implemented consists of a distributed multimedia database, a client-end application and a transport service. The database is made up of three servers: a database manager server, a text and image server, and an audio and video server. The client components are a Graphical User Interface (GUI), a media scheduler and stream synchronization control modules for synchronization purposes, data readers and displayers, and a QoS manager. The transport service consists of a real-time transport service for audio and video and TCP/IP for control messages and text, both running either over Ethernet or ATM networks. The system will be presented in more detail in chapter 3.

1.2.4 What needs to be done?

The Multimedia Research Communications Laboratory (MCRLab) at the University of Ottawa has committed itself to developing and studying a distributed synchronization architecture. To do this required re-engineering the architecture of the integrated system without putting undue stress on the other components of the project.

Namely, it was necessary to find a way for extending synchronization coordination abilities to the servers without forcing the server design teams to radically rethink their work. This required a sound understanding of the issues at hand, which could only be done if a solid framework for understanding the synchronization aspects of a system is provided. This thesis will set out to provide such a framework, study the CITR News-on-Demand system with this framework, propose changes and show how these changes are implemented.

1.3 Thesis outline

The thesis will proceed as follows: chapter 2 will present a framework for studying multimedia synchronization within a distributed application; chapter 3 will analyze the News-on-Demand system with the framework of chapter 2; chapter 4 will show how this information can lead to a safe and effective re-engineering of the system; chapter 5 will discuss the implementation of the new synchronization architecture; the thesis will conclude with chapter 6, which will summarize the points made in the other chapters, compare the old and new synchronization systems, and discuss future research.

1.4 Publications arising from this research

Jarmasz, J. and Georganas, N.D. (1996). *Designing a Distributed Multimedia Synchronization Scheduler,* article submitted to the IEEE Multimedia Systems '97 Conference, Ottawa, Canada.

CHAPTER 2: A FRAMEWORK FOR STUDYING THE SYNCHRONIZATION ASPECTS OF MULTIMEDIA APPLICATIONS

2.1 Some basic synchronization concepts

Simply put, multimedia synchronization deals with the **temporal aspects** of multimedia presentations. A few authors will use the term to talk about any and all compositional aspects of a presentation - i.e. spatial, logical, etc... - but this use of the term is confusing and shall be avoided here. Since multimedia integrates time-independent (text, graphics) and time-dependent (audio, video) media, it is quite obvious that multimedia documents have a temporal as well as a spatial dimension, and this dimension must be coordinated and protected to ensure satisfactory presentations [GA91].

It would seem that to deal with synchronization correctly, all that is needed is to ensure that the right data are played back at the right time. While this is basically true, this actually involves quite a bit of work. Networks and workstations introduce delays; a video clip might arrive on time but it might be jittery - it might play back in an uneven fashion; the captions for a talk might be displayed ahead of the audio they are supposed to accompany. How should these problems be dealt with? Should the errors be prevented before the data arrives at the client application? Should these errors be allowed to occur

on the understanding that the receiving application will be powerful enough to correct them? It is clear that these problems can become quite complex rather quickly, and that it is good to have some kind of framework - some sort or checklist or "problem space„ - within which to examine these issues in an orderly and efficient manner.

A logical separation of the issues suggests itself: the places within the data where synchronization errors occur, the places within the system where the errors occur, and how these errors should be dealt with. These are the "axes„ along which the framework presented in this chapter shall be organized. First, a synchronization taxonomy[SN95] shall be presented, which will illustrate how the degree of abstraction with which the data is viewed helps to understand the types of synchronization phenomena that might be encountered. Next, an "architecture model„ will help us examine where in the system these phenomena occur. Finally, we will examine synchronization strategies to see how errors are dealt with, and how these strategies affect the effectiveness of a synchronization system.

2.2 Classifying synchronization phenomena - a Synchronization Reference Model

As has been stated before, multimedia is about integrating different media - different data types - with different structures and requirements into a single document or presentation. We can deduce from this a few things:

-the type of synchronization errors that can occur will probably vary with the different media that are being presented;

-some of the media will have a relatively complex internal organization; this probably means that the different organizational components of the media will incur different types of errors;

-if we study different media from the point of view of their organizational characteristics, we can probably predict what kinds of errors we are likely to encounter; furthermore, this knowledge should allow us to predict what kinds of errors a system might have to contend with based on the characteristics of the data it deals with.

So, how are the different media organized? Time-independent media, such as text and images, only have an inherent spatial organization - the size of the document, where they go on the screen, and so on. The only temporal information that might be relevant is information that would relate such "objects,, to each other, like in a slide show. This information answers questions such as: Do we show paragraph A before picture B; if so, how long does A stay up before B is displayed? Might they be displayed simultaneously?

Time-dependent media, as their name implies, have an intrinsic temporal, as well as spatial, dimension. They have an internal temporal logic which is absent from time-independent media. A video clip only makes sense if its individual frames are played back in the right sequence and at the right speed. Furthermore, variations in the rate of playback (known as jitter) of the frames is usually a nuisance to the user of the application, and thus undesirable. As with time-independent media, the temporal relation

of time-dependent streams, both to other time-dependent streams and to time-independent objects, must be taken care of. With time-independent media, it is simply a matter of displaying a paragraph of text or an image at a given moment during the playback of a continuous stream - or before, or after. When coordinating different streams, it becomes a little more complicated: the desired relationship of stream A to stream B must be maintained for the duration that both streams are to appear concurrently. Streams have a beginning, a middle and an end, while time-independent objects do not, at least not temporally speaking.

These distinctions are the basis for the Synchronization Reference Model given in [SN95]. A diagram for this model is provided:

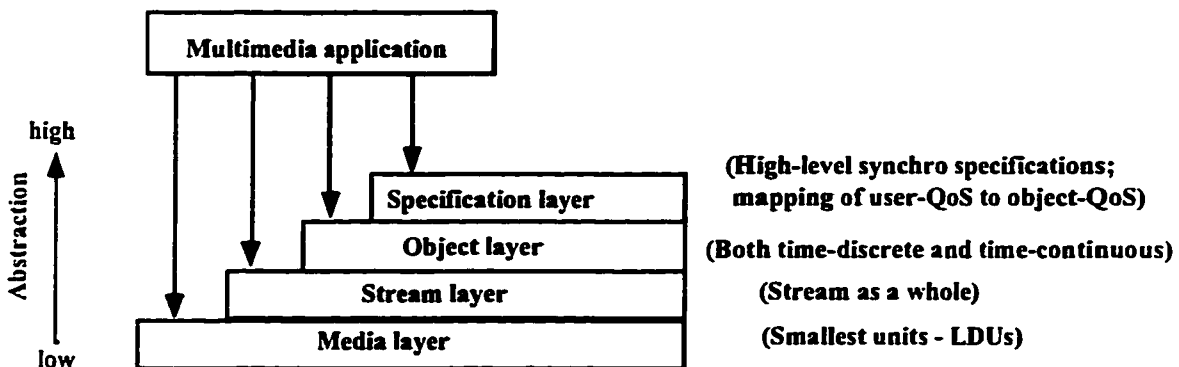

Figure 1: Four Layer Synchronization Reference Model (from [SN95])

The idea behind this model is that, in the same way that an audio clip is made up of successive samples, and a multimedia document is made up of many clips and objects,

synchronization phenomena can be seen as being an assembly of more "basic„ synchronization components. If proper synchronization between objects, both time-independent and time-dependent, is to be maintained, then proper synchronization must also be maintained "inside„ theses components. This information should guide the study or the design of any multimedia system, especially as pertaining to synchronization. The different layers of the model shall now be presented and explained.

2.2.1 The Media Layer

This layer, as well as the next one, deal specifically with time-dependent media. The media layer looks at what goes on inside a single stream. A stream is treated as a sequence of its basic components. What constitutes a "basic component„ can vary from medium to medium, and from application to application. For this reason, it has been given a generic name by Steinmetz: a Logical Data Unit, or LDU. This LDU, whether it's a single frame, an arbitrary sequence of frames, or an arbitrary block of audio samples, is the smallest data unit which is useful to the application.

Anything that is concerned with the internal logic of a stream, be it the retrieval of an LDU, the playback of a single stream or ensuring that the playback rate of a stream is correct, operates at this level. This is the lowest level that has significance as far as data goes. Operations at this level make use of a lower lever - usually the operating system, sometimes special low-level routines in the application - at which the stream as such is

invisible. The media level also provides services for the upper levels - an entity coordinating objects must invoke functionalities at this level to operate.

The synchronization errors occurring at this level (and, by extension, those which are actually due to the operation of lower levels) are known as intra-media or intra-stream errors. They consist of the following: unwanted delays between successive LDUs; variations in the delays between successive LDUs - this is known as jitter; errors in the ordering of LDUs (again due to delays); loss of LDUs.

2.2.2. The Stream Layer

This layer looks at groups of related streams - usually meaning streams being played back in parallel. At this layer, streams are viewed as whole entities; they are the basic operating units of this layer. LDUs are hidden. The notions associated with streams here are overall parameters for streams, such as playback time, playback speed (as a fraction or multiple of the "normal,, playback time), a time-marker inside a stream, or the relationships between streams. This is in fact the layer at which time-dependent streams are coordinated with each other.

Since the internal logic of streams are not explicitly known here, this layer must use the media layer to operate. Stream layer parameters for an individual stream, such as "playback speed equal to normal,, must be translated by the media layer into parameters such as "thirty frames per second,,. The stream layer's functionalities are used by the

layers higher up - those which integrate time-independent and time-dependent media into a whole document.

The synchronization errors which occur at this layer are called inter-stream (or sometimes inter-media) errors. They consists of the delayed (or early) starting of the beginning of a stream; the early or late ending of a stream; the time-skew between streams; the loss of a whole stream.

2.2.3 The Object Layer

This layer views data from the point of view of a whole document. The basic "building blocks,, at this layer are the separate "objects,, which make up a presentation: time-independent objects such as images or text files, and the time-dependent objects under the control of the stream layer, i.e. streams or groups of streams. All media types are integrated here; the difference between types of media are hidden. All objects are seen as having a start time and a stop time here, as well as internal time markers if necessary. This is the layer where the scheduling of a presentation (i.e. the computing of hard start and stop times for the different objects is carried out.

This layer relies on stream layer functionalities to handle groups of streams properly, but handles time-independent objects directly. The "input,, for this layer is the overall temporal specification for a presentation - what is commonly referred to as a scenario, which consists of the temporal relations between objects such as "Start object A

simultaneously with object B„ or "Start object C right after object B is finished„. The "output„ are calls to the stream layer and to the time-independent media players.

The synchronization errors which occur at this layer are called inter-object or inter-media errors. They consist of early or late playback times for objects; the loss of objects.

It should be noted that the object and the stream layer are easily collapsed conceptually. Streams are, basically, objects, and scheduling streams and scheduling objects are pretty much the same thing, and thus would seem natural to treat the scheduling that should occur at the stream layer as occurring at the objects layer. However, one should keep in mind that it is groups of streams - i.e. streams playing in parallel - that should be considered as objects, not the individual streams that make up the group. This is because the temporal evolution of streams in a group are interrelated, and thus only the group forms a temporally atomic object. Computing constraints might make it more practical to see each stream as an object, though, thus indeed collapsing the stream and object layers into one.

2.2.4 The Specification Layer

This layer does not in fact directly correspond to the data that make up a document. Indeed, the highest level of data abstraction in fact happens at the object layer. However, the notion of specification layer was introduced to show that before a schedule

for a presentation is computed, the schedule needs to be somehow specified in general terms - there needs to be some kind of general guidelines for the schedule. This is the scenario that was introduced in the last section.

This layer is most often used by authoring tools. However, this is also a useful abstraction for actual presentations where the schedule for a document must be computed on the fly and take into account user requirements and network conditions. Thus the specification layer is responsible for mapping user requirements and network conditions into a form useable by the object layer. The specifications, in the form of a scenario, provide the conditions of validity for the schedule. A presentation can have many schedules, as long as they fit the original scenario.

Since the specification layer is not actually involved in carrying out the playback of data, there are no synchronization errors to speak of here.

2.3 *Where errors occur - a complete view of the system*

The multimedia systems that are or will be of most relevance are distributed systems - i.e. systems that are shared between many workstations, typically a few servers and many clients. Systems that just play back data from a storage device (e.g. CD-ROMs) are already available and the information they contain cannot be as current and up-to-date as distributed database-oriented services. However, the latter systems are

much more complex, as they involve many different components - different machines, a transport service, many different bits of software. It's quite obvious that delays in data processing and transmission are given may opportunities to occur in such a system.

Each of the "components„ mentioned above have different characteristics, and it is to be expected that some component might induce more errors, or conversely might be able to compensate for errors more easily, than others. To explore this more, the following section will study these concepts: the transport mechanisms, the hardware and operating systems of the workstations, and the applications themselves.

2.3.1 The Transport System

Multimedia applications, especially distributed ones, where the data are located on a machine other than the client workstation, depend on fast and reliable data transport mechanisms for good performance. As we have already seen, the special nature of audio and video data imposes certain synchronization constraints on a presentation; similarly, they also require certain characteristics from the network that connects the clients to the servers.

The complexities of communications networks will not be examined here, as this is beyond the subject of this thesis. Further information on this subject can be found in [SN95] [St94]. However, it is useful to examine what kind of requirements multimedia

applications make on the transport mechanisms they use, how these requirements are met by currently available technology, and how all of this relates to synchronization.

The desired qualities of a communications network depend on the type of data or messages that are going to be transmitted over it. The interactions between the user or users and the multimedia service being provided determine the required broadcast format: some applications have multiple users communicating with one server (many unicast connections), others have one server communicating the same data to many clients at once (broadcasting) and others have many clients interacting, possibly through a central server (multicasting or conferencing applications). Multimedia applications make use of the following types of message or data formats:

-multimedia data: this includes time-dependent and time-independent media. Time-dependent media generally require a very high bandwidth (especially video, which can require a few megabits per second), fast data forwarding (i.e. low waiting times for starting to receive data) and soft service guarantees (i.e. the ability to guarantee a given throughput or other such parameters, but without impairing the end-user application if acceptable violations of the guarantee occur) [SN95]. In other words, the priority providing and guaranteeing time performance. With time-independent media, however, the priority is reliability. Text and graphics, however, need considerably less bandwidth than audio or video and do not have any intrinsic temporal semantics; it thus costs relatively little to ensure error-free transmission for these media.

-control and command messages: the clients and the servers must have a reliable way of transmitting messages and requests. The requirements for this type of communications are basically the same as for time-independent media (extremely high reliability, low bandwidth), except possibly when control of the data streams is desired (i.e. some kind of feedback from the client to the server), but this is more a question of network protocols than of multimedia applications.

We can see from the above discussion that multimedia basically have two kind of requirements for communications services: on the one hand, they need services with very good time-performance (i.e. high-bandwidth, low jitter, connection-oriented) characteristics and relatively good reliability for time-dependent media, as these media often cannot be retransmitted in case of loss or error[1]; on the other, they also need services with very high reliability but which don't necessarily have extremely good time-performance characteristics for time-independent media and control messages (although long texts and images do need much more bandwidth than control messages.

The network architecture that is typically used for this last type of communications is the traditional TCP/IP network. It is a packet-oriented relatively low-bandwidth, reliable network which provides no QoS-type service guarantees. While it is adequate for text and acceptable for graphics, it was very quickly realized that it was inadequate for transmitting audio and video in a satisfactory way. The great hope for this

[1] While many types of time-dependent data can suffer losses or errors during transmission without seriously compromizing performance, some formats high-compression, such as MPEG2, can't tolerate very many transsion errors and thus require a very reliable transport service.

type of communications in the ATM (Asynchronous Transfer Mode) architecture, which allows extremely high bandwidth, excellent reliability and a variety of services modes which enable it to provide a wide range of service guarantees. However, this technology is still fairly new and quite expensive. As a result, protocols which use some part of the TCP/IP infrastructure to transmit audio and video in "real time" have been developed [SN95]. The UDP (User Datagram protocol), for instance, replaces TCP over IP. It provides only a basic service of multiplexing and checksumming, so loss of data does not force time-consuming retransmissions, as in TCP. It can be used as an unreliable but quasi-real time transport medium. Other examples of protocols with multimedia applicability can be found in [SN95]. It should be noted that research on multicasting capabilities is still very much under way; one of the most interesting projects is the MBONE system which provides basic multicasting functionalities for the IP network architecture [SN95]

2.3.2 Hardware and Operating Systems

Operating systems provide the software interface that allow applications to use a computer's physical resources. Thus, to the application, the operating system and the hardware appear as one. No matter how good the hardware is, the application can only use what the operating system gives it, unless a lot of time is spend creating new device drivers and file access schemes. Similarly, what the operating system can ultimately provide is bounded by the hardware. Each one limits the other.

As we have seen earlier, multimedia applications have specific temporal requirements, and therefore require certain functionalities from the hardware/operating system team. They need to be able to do real-time process scheduling [GA91], real-time resource allocation, device management, data retrieval in real time as they are presented to the user and accurate timing services [SN95]. Real time issues are also discussed in [SN95]; simply put, a real-time process or service produces results right on time. If the requirements outlined above are not met, the application will not be able to ensure correct synchronization of the presentation.

Surveys of popular operating systems, such as UNIX, Windows and MacOS [BL91] [Bur94] [SN95], show that such systems do not provide adequate real-time performance or timing services for multimedia applications. There are many experimental or research real-time operating systems which might be suitable [Bur94] [LMM94], but the fact remains that applications that are expected to have any kind of wide-spread or commercial success must be made to work on the most common operating systems [SN95].

The solution in this case is to bypass parts of the operating system's functionalities and to replace them with more suitable services. A software library which provides a real-time environment within the standard operating system is required for this purpose. One such library is the Real Time Threads package from the University of British Columbia [FHM+95]. The problem with this approach is that the application

must still run on top of the operating system - the application must still run on a CPU which is handling other non-real-time processes, and the real-time performance provided by the software library is somewhat compromised.

At any rate, any multimedia system must be able to provide real-time functionalities to the application, either through a suitable operating system or specialized software libraries running on high-performance hardware.

2.3.3 The application

The application is what really does all the work. Most research on multimedia focuses on applications. A distributed multimedia application must be able to present time-dependent and time-independent media while reading data off of the transport service and ensuring the temporal integrity of the presentation. The application must also be responsive to the user's needs. These considerations allow us to provide the following requirements common to all multimedia applications: integration of different media; guaranteeing the quality of the presentation, both at the "spatial„ (e.g. format for audio, resolution or picture size for video, etc.) and at the temporal (i.e. synchronization) level; allowing some level of user interactivity.

It goes without saying that a multimedia application has to integrate different media - particularly time-dependent and time-independent media. Without this there is no multimedia. It is enough to say here that with the new technologies available to the

average personal computer or workstation user, the challenge is not simply presenting many different media at once anymore, but rather doing it in such a way that the overall quality of the presentation is acceptable to the user.

The quality of a presentation is determined mainly by the author of a presentation, and also by the user to a certain extent. It includes such things as choosing an encoding format for audio, or choosing the number of colour levels for an image. Often the user is given the option of choosing between different qualities for a given aspect of the presentation - e.g. a video could be displayed at five, 15, or 30 frames per second. These choices are the user's Quality of Service (QoS) requirements. The QoS paradigm is a concept that was adapted from computer networks to multimedia [Bur94], and is basically a way of keeping track of the quality requirements for a presentation throughout the system, and of mapping the requirements between the different components of the system. An application must ensure that these requirements are met, either by trying to meet them at the server end and hoping for the best, or by compensating for any deviations for the requirements at the client end whenever possible, or, ideally, by doing both these things.

Guaranteeing the "spatial„ quality of a presentation is basically a question of ensuring that the required resources (CPU time, memory, devices) are available and reserved by the application [SN95], and that the data arrive without distortions to the user. Guaranteeing the temporal quality of a presentation is achieved by using a good synchronization strategy. This aspect was discussed in an earlier section. To recap, a

good synchronization strategy includes both preventive (i.e. scheduling) and corrective measures, at all levels of data granularity (media, stream and object layers) and in as many component of the system as possible (but mainly at the server and the client).

The application must be able to interact with the user to a certain extent. This is obviously necessary to allow the user to control the presentation's temporal unfolding - i.e. starting the presentation, stopping it, rewinding it, etc. - but also to allow the user to determine the desired QoS level and negotiate with the application if a desired level is unavailable. Both of these types of interaction have effects on synchronization: QoS negotiation determines the parameters which are to be respected during synchronization, while controlling the presentation determines the actual time and sequencing of the presentation.

2.3.4 Review

Once the desired qualities of the presentation (i.e. synchronization parameters) have been decided, the server applications try to meet these parameters (preventive synchronization), and the client applications try to restore any deviations from these parameters (corrective synchronization) which have been incurred due to the performance of the underlying systems - transport, operating, and hardware. Of course, any design inefficiencies in the application will either introduce more synchronization errors into the data (e.g. if the priorities of concurrent processes are not properly assigned), or simply make the correction of errors more difficult.

2.4 *Synchronization methods: prevention and correction*

The basic methods of ensuring correct synchronization of multimedia data have been alluded to in the previous sections. They are presented in more detail here.

2.4.1 Prevention

"An ounce of prevention is worth a pound of cure,,, as the old saying goes. This is true for multimedia systems as well. It is a fact of life that data will arrive with synchronization errors at the client end, but there are steps that can be taken to minimize these errors and reduce the amount of work that has to be done to compensate for them.

One aspect of prevention is ensuring that the "infrastructure,, which the application uses -i.e. the operating system, the transport service and the hardware - are as efficient as possible. This has been discussed previously. The idea is to use high-performance technologies which allow a flexible approach to timing issues and give a certain amount of guarantees about quality of service. These aspects are often not dealt with directly by the designers of an application - more often than not, they simply make a choice about which system to use.

Another aspect of prevention, as has already been mentioned, is scheduling. Knowledge about the characteristics of, and the actual conditions prevailing in, the system which underlies the application can be used to determine when data should be retrieved from the storage medium at a server and sent to the client. For example, if we want to display video V at time t, and we know that it takes f seconds to fetch it off the disk, n second to send it over the network, and d seconds to decode it, the video should be sent at time

$$ts = t-f-n-d.$$

In other words, V should be sent f+n+d seconds before it is to be displayed. This approach to scheduling is explained in [LKG94.2] [Li94]. This is the basis for the scheduling algorithm to be used in the CITR News-on-Demand system.

2.4.2 Correction

No matter how sophisticated the prevention strategy is, errors will occur. It is therefore essential that a multimedia application have some kind of strategy to compensate for these errors.

One such strategy is the Synchronization Stream Protocol (SSP) developed at the MCRLab [LLG94] [LLBG96]. The basic idea is this: if a presentation object arrives late, the other objects in the presentation should be delayed by the application so that the relative times of the objects are once again correct, as long as this doesn't violate the original scenario for the presentation. If the elements inside a stream are being delayed,

then synchronization with other concurrent streams should be kept by skipping selected parts of the delayed stream (this is particularly applicable to video - the human visual system is less sensitive to gaps in data than the auditory system [SN95]). This allows the presentation to appear synchronized to the user, even though errors have occurred during the transmission of data. This is the correction algorithm to be used in the CITR News-on-Demand system.

2.5 *Putting it all together - the three axes*

From the preceding sections, we can say that the synchronization strategy or mechanism of any multimedia system can be studied along three axes: the media granularity axis (which classifies synchronization phenomena), the origin axis (which classifies the system components where these phenomena occur), and the synchronization method axis (which classifies the methods used in dealing with these phenomena).

This framework is designed to make a qualitative assessment of the synchronization aspects of a multimedia system quickly and easily. The three axes divide the conceptual space of the framework into discrete regions. By placing the components of a system in these regions, the existence of synchronization facilities and properties in this system can be represented. As an example, consider a system with one data server (DS) which performs intra-stream scheduling, a database server (DBS), a transport service

based on an ATM network (ATM), and a client (CL) running an application, consisting of a data reader (DR) and a data scheduler (DSch) which collaborate to correct synchronization errors at the object and media layers based on an object layer presentation schedule, running over a UNIX platform (UNIX). Figure 2 illustrates how this system can be represented with the framework.

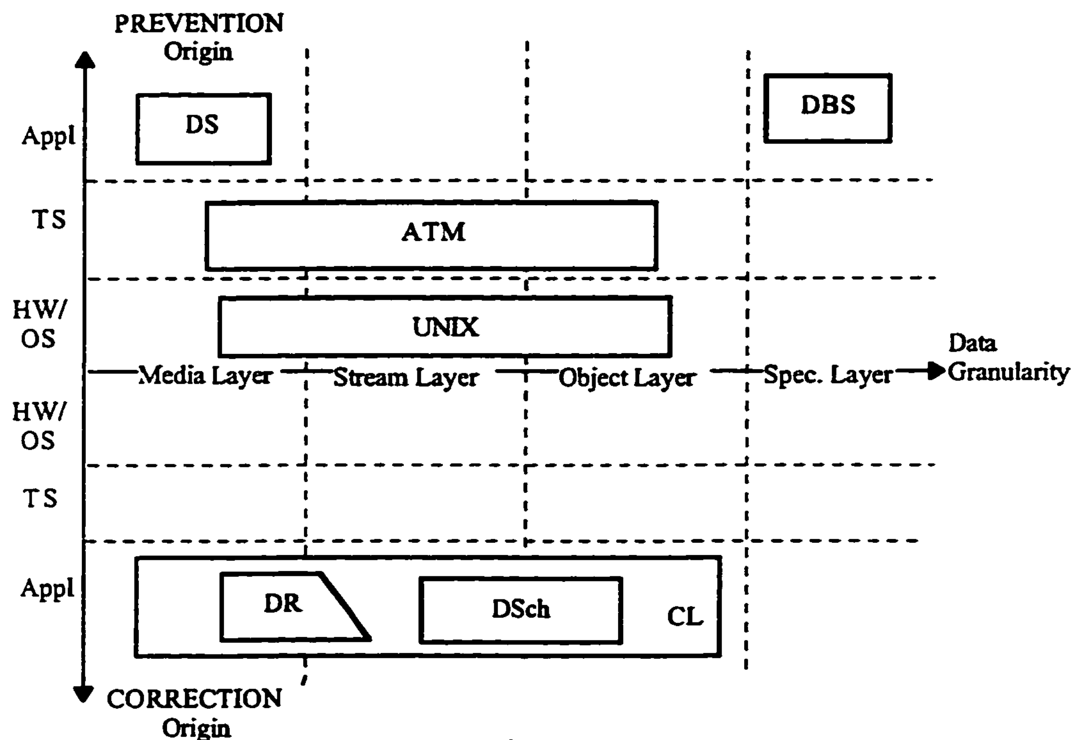

Figure 2: Using the three-axis framework to represent a multimedia system

Since it is not practical to represent a three-dimensional space on a two-dimensional medium, the upper half of the graph represents objects along the prevention axis, while the lower half represents objects along the correction axis. Components represented by rectangular boxes indicate that the component provides synchronization

facilities in a given region or regions; while it is assumed that such a component is adequate for its responsibilities, this is not specified in this representation. If the components has known properties (such as an ATM network), then it can be assumed that the component has the potential to perform adequately, at the very least. Components represented by boxes with sloping edges indicate that the component is known to provide poor or partial synchronization facilities. Components spanning more than one region can be assumed to provide adequate (or undetermined) facilities in one region and partial ones in another, such as the DR component in figure 2. Useful information about the system can easily be obtained from this figure. It clearly shows that the system performs only media layer scheduling, so, despite the high-performance ATM network, this system will not be good at preventing errors. Furthermore, the UNIX operating system will certainly further impair the system's performance. The corrective functionalities of the system appear to be quite comprehensive, as they operate at every data level (note: the presentation schedule is only used in the correction stage). While no quantitative performance parameters can be determined from this diagram, we can easily tell certain key aspects of multimedia synchronization are not properly addressed by this system - namely delivery scheduling at the stream and object layers, and the performance of the operating system.

The axes used in Figure 1 have only a few divisions each. They can be further subdivided if necessary. For instance, a Scheduling subdivision could be included in the Prevention part of the Strategy axis; the Application section could be subdivided divided into Client and Server sections; Hardware and Operating System could be separated.

These refinements allow the graph to be more meaningful and to convey more precise information about the synchronization characteristics of the system.

The next chapter will show how the CITR News-on-Demand system can be analyzed using this framework. The following chapter will also use this framework to design a distributed scheduler.

CHAPTER 3: ANALYSIS OF THE ARCHITECTURE OF THE CURRENT NEWS-ON-DEMAND PROTOTYPE

3.1 Overview of the application

The CITR News-on-Demand prototype is a system meant to provide users with rapid access to an interactive multimedia news database from their personal computers or workstations over high-speed communications networks. At a very general level, the system is a user-end application communicating with some kind of multimedia news database via a transport service.

The user interacts with the system through a GUI. The GUI allows the user to select an article for viewing and to choose a certain quality level for the presentation, out of a number of options provided by the system.. Once these choices are made, the user controls the flow of the presentation with VCR-style commands (i.e. play, stop, fast-forward, rewind, pause).

The user's requests are forwarded by the GUI to a coordinating module in the client called a scheduler. The scheduler retrieves the scenario for the article selected by the user from the database and computes a presentation schedule based on the scenario and user requirements. The user's choice of QoS parameters (which are derived by the application from the overall quality level chosen by the user), such as a given frame rate for video, or a certain quality of audio, determines which variants of the presentation will

be used [BKH+95]. The user also decides when the presentation is to start, thus determining the absolute times for the schedule.

The scheduler requests the components of the article from the database and initializes the modules which will read data from the network and present them to the user. One of these modules is the Media Synchronization Controller (MSC). Each data connection has its own MSC. The MSC monitors a data connection for any variations on the rate of received data, and informs the scheduler of any delays with regards to the schedule. The scheduler will reschedule the presentation times of any component as necessary, as long as the new times do not violate the original scenario. Should this happen, the application try to replace the presentation objects causing the excessive delays with another version of the objects that have better QoS parameters (so-called QoS renegotiation [BKH+95]). If this is not possible, the presentation is aborted.

The database for this system stores audio, video, text and graphical data, as well as meta-information about the articles. The user only sees the GUI, and thus from his point of view there is only one large multimedia database. In fact, the database is made up of many server applications, each of which is suited to particular media types. There is an overall database server for meta-information, and servers for time-dependent, as well as for time-independent, media. Each server provides an interface between itself and the scheduler; these interfaces allow the scheduler to set up data connections between the servers and the MSCs and to initiate or terminate the sending of data by the servers, as well as to specify the parameters for the transfer of data.

The quality of the presentation is determined by the QoS parameters, which are determined from the user's choices and the parameters provided by the database. The negotiation and transmission of these parameters is handled by the QoS manager. Whereas the user is only presented with a number of choices concerning the overall quality of the presentation, the QoS manager must map these choices into parameters that are useful to the database and determine the exact value of these parameters, taking into account both what the servers can provide (based on available data and system load) and what the user wants. If the user's choice cannot be met, the QoS manager presents the user with alternatives which are compromises between what was originally requested and what can actually be provided. For more details concerning the QoS manager system, see [BKH+95].

The communications between the user's end of the application and the servers is ensured by the transport service, which provides service modes suitable for the different types of data which are used by the system. Not only must control and data communications between the user and the database be ensured, but communications between the different servers making up the database must also be provided.

The overall functioning of the system is illustrated in figure 3 (from [Geo96]). The following section will examine in more detail how the different components of the system have been implemented.

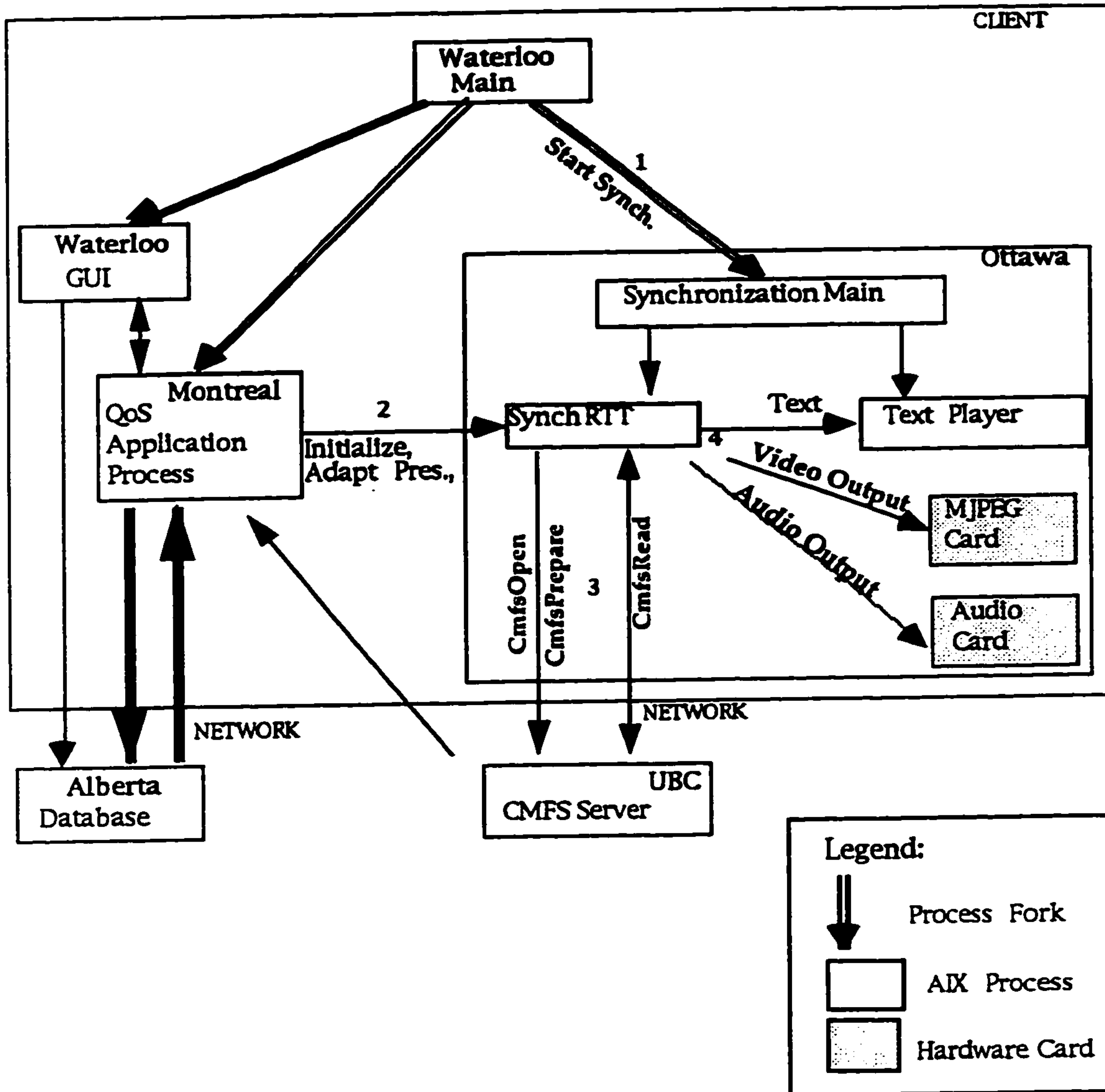

Figure 3: Architecture of Prototype with Centralized Scheduler (from [Geo96])

3.2 Implementation of the architecture

From the previous section, it is clear that the News-on-Demand system is a distributed application which has a client-server architecture, with a distributed database -

i.e. the different media are distributed among many different servers - and a network to connect the client and the servers.

How was each of these general components implemented? As we have seen earlier, the issues that such an application must deal with are: streams of time-dependent data as well as time-discrete data and control messages; data retrieval scheduling; a correction mechanism for dealing with synchronization errors at the client end; user interactivity; a fast and reliable transport service; a reliable real-time environment; guaranteeing quality of service throughout the application. All of these concerns shape the different components of the application.

The distributed multimedia database is made up of many separate data servers. In its most basic form, is comprised of three servers: a multimedia database server, a text and image server, and an audio and video server. The database server stores all the meta-information about the articles which are stored among the different servers of the database. This is the information that tells the application how to present the data - in other words, it's such things as the scenario for an article and information about the components of the article, such as duration, size, media format, etc. The text and image server stores the time-independent data. Both of these servers were implemented by the research team at the University of Alberta [LGOS96], [OSEV95]. The audio and video server stores the time-dependent media, which have different synchronization requirements than the time-independent media, as was discussed earlier. This server was implemented by the Continuous Media File Server (CMFS) team at the University of British Columbia.

The client application takes care of user interaction, scheduling, QoS management, corrective synchronization, and displaying data. The synchronization aspects were implemented by the team from the MCRLab at the University of Ottawa [BG96]. The user interface and the display functionalities are the responsibility of the integration team at the University of Waterloo. The synchronization components consist of: a central scheduler, which starts data reader entities, MSCs, which monitor incoming data for synchronization errors, and display units. The scheduler and the MSCs interact to implement the SSP as discussed in sections 2.4.2 and 3.1. The MSCs open connections to the servers and request data, while the scheduler determines when the data readers should start reading data, based on when the user presses the "Play" button on the GUI. QoS management is provided by the team from the University of Montreal. The main component of the QoS Manager is at the client side (there are to be local QoS Managers located on the data servers), which needs to communicate with the other components of the system via the RPC paradigm [BKH+95]. The client communicates with the servers via Application Programming Interfaces (APIs) which are provided by the teams which developed the servers.

The transport service consists of the physical network, and the protocols to be used over it. The News-on-Demand application is designed to work on a variety of network architectures, although the target architecture is an ATM network. To ensure proper delivery of time-dependent data streams, the CMFS team has integrated a continuous media transport service, called Media Transport (MT), into their server API

[Me96]. This service uses the UDP protocol to transport time-dependent data in a way that is transparent to the user, and to the application. Control messages between the client and the CMFS use the Send/Receive/Reply Remote procedure Call (RPC) mechanism [MHN96]. Information about the communications between the database server and the client, and the text and image server and the client, are still unavailable, but since they will involve mainly querying the database server [OSEV95], they shall probably use the Remote Procedure Call (RPC) model over TCP/IP as well.

For the purposes of synchronization, QoS parameters can be assimilated to meta-information, and transport protocols and network architectures are only relevant in that they induce synchronization errors. Therefore, the analysis which will follow will focus mainly on the client and the server ends of the system. Figure 4 highlights the synchronization components of the prototype.

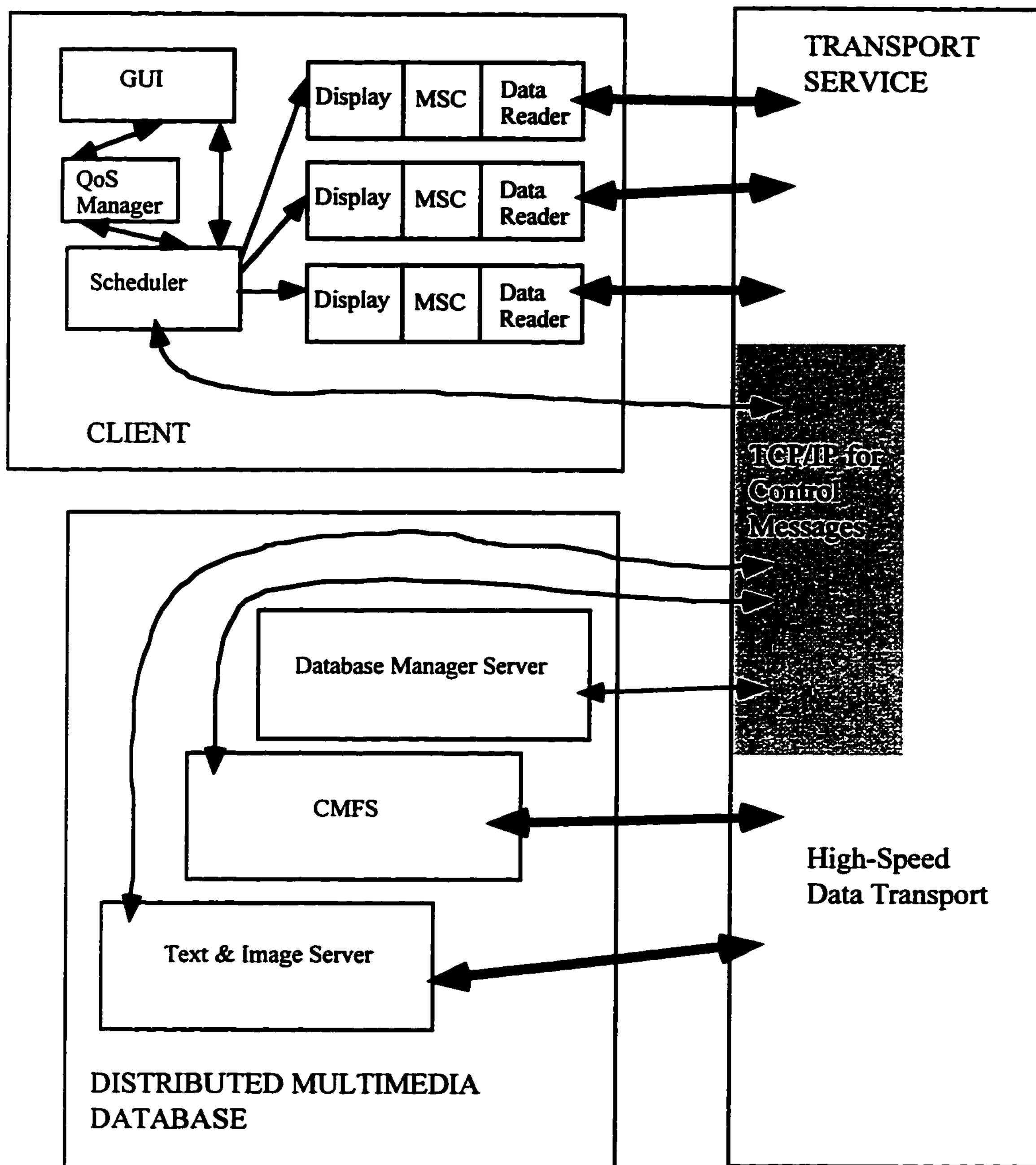

Figure 4: Communications Between the Components of the System

The prototype was implemented using the C and C++ programming languages. The real-time environment for the client and the servers is provided by the Real Time

Threads (RTT) package created at UBC [FHM+95], which requires the C programming language. The system runs on IBM RS6000 machines running the AIX 3.5.2 operating system, and can use either ATM or Ethernet networks.

3.3 *Preventive strategy*

3.3.1 Server Applications

The server application most relevant to synchronization (and the only one available for study) is the CMFS [MHN96]. The CMFS provides a client application with the ability to access data streams at guaranteed high speeds while respecting the data's real-time semantics and QoS requirements. The CMFS meets these goals by using the MT transport service mentioned earlier and by providing media-layer scheduling (referred to as "disk scheduling,, in [MHN96]). Which streams are accessed and when is entirely up to the application using the CMFS - no stream- or object-layer scheduling is provided.

The APIs for the Text and Image server and the Database server (likely to be implemented as one server [OSEV95]) was not known to the author at the time of this writing. However, what is known is that the APIs for these servers will allow querying of the database using the SQL database language [LGOS96] [OSEV95]. This only involves sending relatively short text messages between the client and the servers; therefore, it is

likely that these servers will use the TCP/IP protocol stack. It is also likely that the Text and Image server provide intra-object scheduling at most, if any. This is because this server, like the CMFS, is designed to be as flexible and general as possible, leaving any inter-database entry semantics entirely in the hands of the user [OSEV95].

3.3.2 Client Application

Since the servers provide no scheduling above the media layer, this aspect must be taken care of by the client application. The scheduling theories developed at the University of Ottawa [LKG94.1] [LKG94.2] are quite sophisticated; however, the article format currently being used to test the application - three streams, one of video, one of audio, one of text captions accompanying the audio, all of equal length and all starting simultaneously - is exceedingly simple. Consequently, stream- and object-layer scheduling has been neglected in the current, centralized scheduler architecture. The structure of the scenario being used in the current synchronization code bears this out:

```
#ifndef SCENARIO_INCLUDED
#define SCENARIO_INCLUDED

#include "thread_ids.h"
#include "constants.h"
#include "p_time.h"

/*********************************************************************/
/* Segments structure contains all information about each individual    */
/* segments for a stream. The stream can be anything, ie MJPEG, AUDIO,..) */
/*********************************************************************/
struct Segments {
   /* These next vars. will be retrieved from the database */
   int         media_type;
   int         segment_number;
   int         segment_length;          /* bytes */
   p_time      segment_duration;     /* in microseconds*/
   int         fps;                  /* Frames Per Second */
   int         rate;                 /* audio rate ie. 8000, 22050,...*/
   p_time      time_start;           /* start segment (in microseconds) */
   struct Segments                   /* Other segments this one depends on
                                        in temporal relationship. */
               *depends_on_segments[MAX_NUM_MEDIA];
   struct Segments                   /* The next segment to be played after
                                        this one. */
               *next;
```

```
    /* These next vars. will be calculated based on the values given */
    /* by the QoS module                                              */
    p_time      min_late;       /* min. amt to change delta late */
    p_time      max_late;       /* abort or drop */

    /* These next vars. will be filled in dynamically by Sync. code */
    p_time      arrived;        /* time segment data arrived */
    p_time      late_arrive;    /* late arrival (calculation) */
    p_time      time_played;
    p_time      actual_late;    /* calculation */
    int         set_delta_late_flag;    /* first one to notice previous segment
                                           was late, changes delta_late & flag*/
    int         datalost_flag; /* Whether data was lost in the segment.
                       TEMP: this should be whether lost in a frame, not seg.*/
};
typedef struct Segments Segments;

/*******************************************************/
/* Stream contains all the information about a media stream */
/*******************************************************/
typedef struct {
    /* These next vars. will be retrieved from the database */
    int         media_type;
    p_time      duration;               /* in seconds */
    int         num_segments;
    Segments    *first_segment; /* First segment in media */

    /* These next vars. will be filled in dynamically by Sync. code */
    thread_ids  *thread_ids;   /* each stream knows other thread ids */
} Stream;

/*******************************************/
/* The Scenario of the multimedia news article */
/*******************************************/
typedef struct {
    /* These next vars. will be retrieved from the database */
    int         num_media;      /* Number of parallel media's in article */
    Stream      *Stream[MAX_NUM_MEDIA];      /* 1 for each stream */

    /* These next set of vars. will be passed by the QoS module */
    long int    *skews;         /* list of inter-media skew
                                   synchronization values.  Not
                                   implemented yet. */

    /* These next vars. will be filled in dynamically by Sync. code */
    p_time      epoch;          /* relative start */
    p_time      delta_late;     /* Delta late for entire presentation */
} Scenario;

#endif  /* SCENARIO_INCLUDED */
```

(File scenario.h by J. Brinskelle)

The scenario is made up of individual streams (up to a maximum of MAX_NUM_MEDIA), which in turn are made up of a series of segments. Clearly, the streams which make up the scenario do not have individual start times - only the overall scenario does. This means that all three streams are requested at the same time by the client (this responsibility is carried out by the data reader modules). Furthermore, no time-independent media are included. Text is treated as a stream of captions to be

synchronized with the other streams. There is, however, evidence of media-layer scheduling - as the segments structure shows. These segments are analogous to the LDUs seen in section 2.2.2 and have temporal information associated with them.

3.3.3 The Infrastructure

The RTT environment makes up for the unsuitability of UNIX-type operating systems (such as AIX) for multimedia, and thus makes real-time performance possible to a certain extent. The MT service provided in the RTT package also allows a certain degree of real-time data transport for time-sensitive streams. Obviously, the system achieves optimum data transport when an ATM network is used instead of the more common Ethernet.

3.4 Corrective Strategy

3.4.1 The Server Applications

Servers are not usually involved in the correction of synchronization errors, and the News-on-Demand system is no exception. Correction can only really take place once data are received, therefore, it can only really be carried out at the client. However, some thought has been given to providing the system with a feedback mechanism between the

MSCs and the servers, thereby involving the servers in corrective operations to a certain extent. This might be the object of future research.

3.4.2 The Client Application

The News-on-Demand prototype uses the SSP to correct errors at the client side. This protocol, explained in [LLG94] and [LLBG96], is carried out jointly by the MSCs and the scheduler. The scheduler takes care or stream-layer correction - when a whole stream is late, it tells the MSCs to delay reading the other streams. The MSCs deal with media-layer correction by smoothing out jittery streams, achieved by correcting delays between LDUs, and by dropping certain LDUs as they arrive too late. The algorithms of the SSP are based on studies of the human perception system done at IBM Germany [SE93] [SN95]. The most important result of these studies for the News-on-Demand application is that, as long as the time-skew between audio and video streams is less that 80 ms either way, the viewer will not perceive any lack of synchrony, and therefore any skews that fall within this range can be ignored. The SSP will be examined in more detail in the next chapter.

3.4.3 The Infrastructure

The infrastructure has a limited contribution to corrective operations. The use of error-correction codes in data compression can compensate for a certain amount of

distortions of the data, and critical control messages can be repeated if they are not received correctly. However, these are not true synchronization error correction operations. The retransmission of data is in fact a synchronization liability, and is therefore avoided for time-sensitive data. Error correction codes can avoid loss of data, which would be detrimental to data synchrony, but they do not provide actual synchronization facilities. For all intents and purposes, synchronization error correction is located in the client application.

3.5 Evaluation of the synchronization capabilities of the current prototype

The preceding analysis is summarized in figure 4. From the figure, we can see that the current prototype provides: a correction mechanism on all relevant synchronization layers, which is located at the client; a media layer scheduling algorithm in the CMFS servers and at the client, as well as rudimentary stream layer scheduling at the client; an infrastructure which provides real-time facilities for data transport and for the applications themselves.

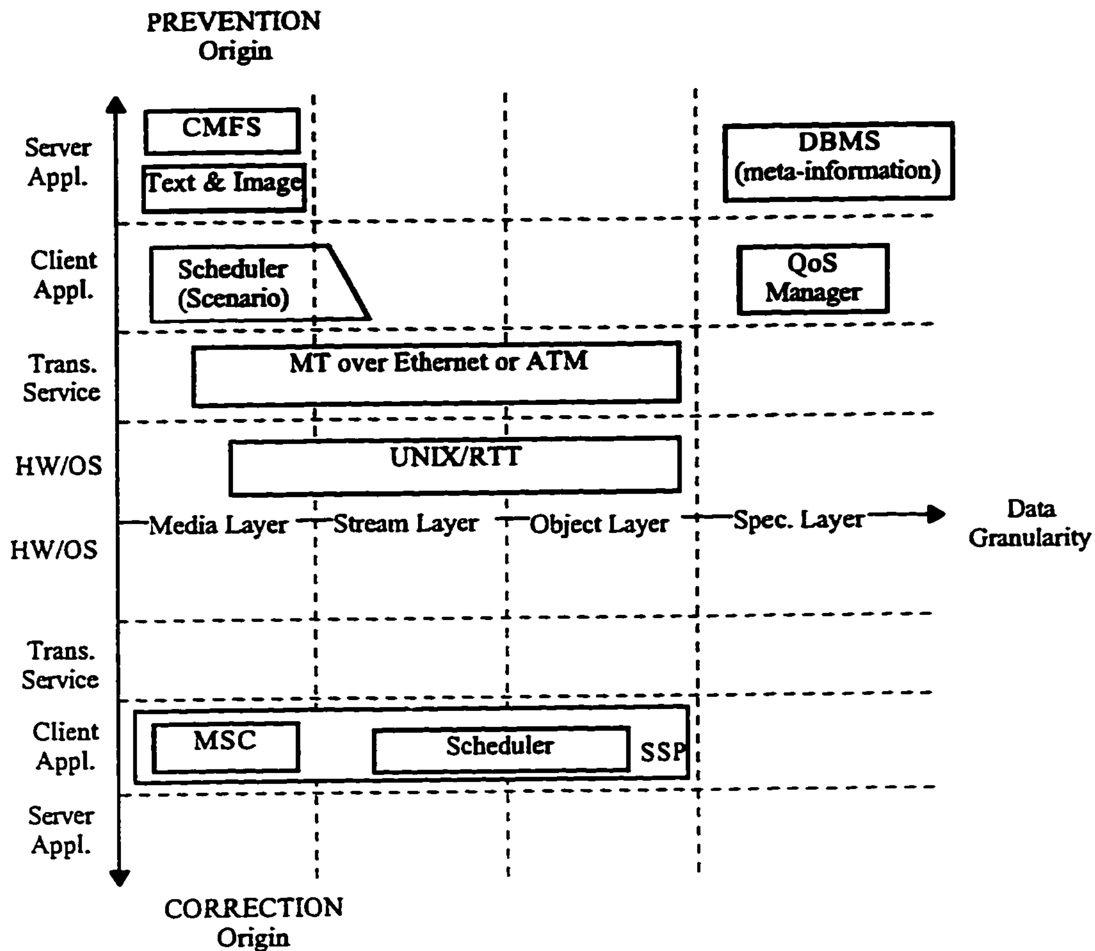

Figure 5: Analysis of the Centralized Scheduler System Architecture

It is quite easy to see from the figure that most of the components of the prototype are involved in some way in prevention operations. However, at the application level - i.e., when it comes to scheduling - there are hardly any components operating at the stream and object layers. The only such component is the client scheduler, which provides inadequate stream layer scheduling. The CMFS, as it is designed, cannot provide any scheduling facilities above the media layer. This should not

be a problem for very simple scenarios - e.g. displaying only a video clip with a synchronized audio clip - but it is insufficient for more complex scenarios which have streams or objects which start after the rest of the presentation has started. This type of document would not be properly displayed by the current prototype. The following chapter will show how to solve this problem with a distributed scheduler.

CHAPTER 4: RE-ENGINEERING THE ARCHITECTURE OF THE CENTRALIZED-SCHEDULER PROTOTYPE

4.1 The MCRLab's Scheduling Strategy - A Distributed Scheduler

The MCRLab has conducted considerable research into the issue of scheduling [LKG94.1] [LKG94.2] [LLG94] [Li94]. The result of this work is an original synchronization specification method - the Time Flow Graph (TFG) - and a distributed scheduler architecture for multimedia presentational systems. This chapter will first present the TFG and the distributed scheduler, and show how these concepts avoid the shortcomings of the centralized scheduler model which was presented earlier. Then we will see how the transition from the centralized model to the distributed one can be made most effectively.

4.1.1 Overview of the distributed model

The distributed scheduler model was originally presented in [LLG94]. The basic premise is that since the servers are responsible for sending data to the client, they are in the best position to determine the most optimal times for transmitting data, which is the essence of scheduling. The basic components and algorithm of this model are presented in figure 6.

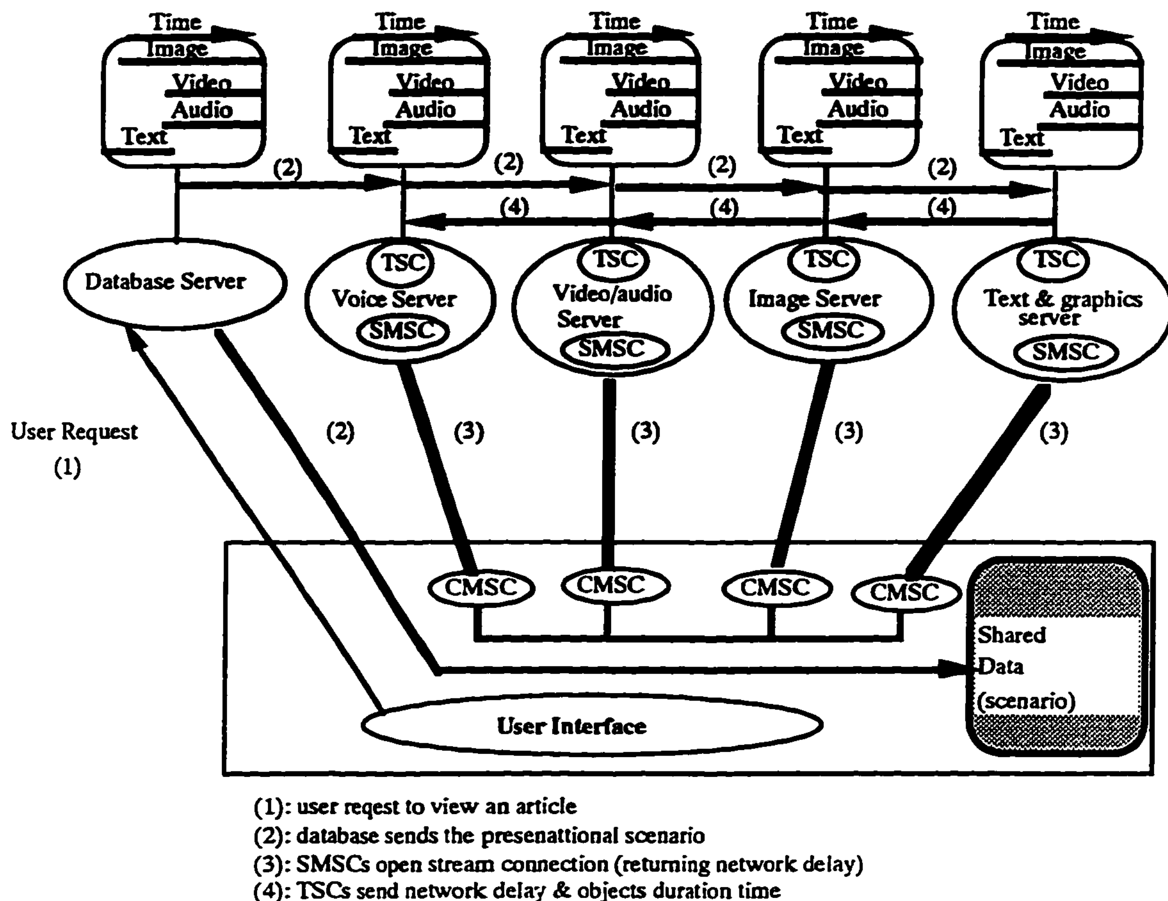

Figure 6: A Distributed Scheduling Algorithm (from [LLG94])

The user requests an article for viewing. The request is forwarded by the user interface to the database server, which then sends the scenario for the article to the client application. However, the scenario is also sent to the servers. The servers then initiate data connections to the client application, based on the information contained in the scenario, namely how many objects and what media types make up the article. The connections are in fact jointly managed by two MSCs, one at either end of the connection. The Server MSCs (SMSCs) open the connections to the Client MSCs (CMSCs), which return the network delay on each connection to the SMSCs. The servers then compute a delivery schedule for the presentation based on the scenario and the network delays

obtained from the client. The process responsible for carrying out the scheduling at the servers is called the Temporal Scheduler Controller (TSC). There is one TSC per server. Each TSC broadcasts the delay between its server and the client, and the duration of its objects, to the other TSCs involved in the presentation. The TSCs therefore jointly determine the delivery schedule.

While these processes are performed at the servers, the scheduler at the client is computing a presentation schedule. When the user requests that the presentation begin, the SMSCs are notified and start sending data. The presentation itself proceeds as it does in the system with a centralized scheduler: the client reads data from the network and displays it to the user. Synchronization errors are detected and corrected using the SSP as discussed in chapter 3.

4.1.2 The algorithms behind the scheduler

The distributed scheduler architecture evolved out of the scheduling algorithms developed at the MCRLab [LKG94.1] [LKG94.2]. The basic philosophy of these algorithms is as follows: the scenario of a multimedia presentation must be specified in a flexible yet comprehensive way. This specification must allow for a certain amount of contingencies during presentation, such as user interactions or the inevitable synchronization errors; however, it must also contain enough information to allow the calculation of the actual times at which the components document are to be transmitted to

the client - i.e. the delivery schedule - and presented to the user - i.e. the presentation schedule.

We have already seen the delivery and the presentation schedules. The notion of a scenario has also been introduced. The transition from the scenario - which is basically a text document, a script for the presentation - to the schedules - real times - is accomplished using the Time Flow Graph (TFG).

The TFG is a mathematical model for specifying the temporal relationships between specified in the scenario. It bears a cosmetic resemblance to the Object Composition Petri Nets (OCPN) discussed in [LG90] [SN95]. OCPNs are Petri Nets extended with duration specifications. The strength of the TFG lies in the fact that it is interval based, and in its ability to represent time intervals of indeterminate length. Being interval based, the TFG represents the duration of events. Events can be multimedia objects or interactions with the user. The latter, whose duration are typically impossible to predict, are usually difficult to model using other specification methods, because these methods often need the times at which events occur to describe the presentation (so called time point-based models) or they make no provisions for events of unknown length (as is the case with OCPNs). However, the TFG allows a presentation to be accurately and efficiently described even if the duration of the events cannot be determined until they actually occur. To illustrate this point, a simple TFG and a simple OCPN are used to illustrate the same scenario: objects A and B, both of the same duration, start together; object C starts immediately after both A and B end.

The TFG model of this scenario is illustrated in figure 7.

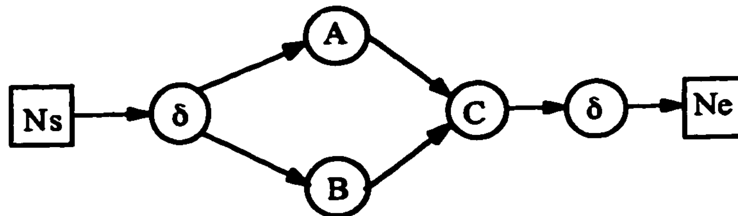

Figure 7: TFG model of a simple scenario

The boxes and circles represent nodes. The object interval nodes, shown here as the circles, represent the interval of time corresponding to the duration of the object. A, B and C are the objects in the scenario. The δ represents an interval of indeterminate duration. The transit nodes, shown as boxes, have zero duration. Ns is the start node of this scenario, while Ne is the end node. The directed edges (the arrows) show which object follows which, and have no time interval associated with them. This graph can be interpreted as "After some unknown time δ, objects A and B start at the same time, and end at the same time. When both A and B end, object C starts; some time after C end, the scenario ends.„ This is equivalent to the scenario given earlier. Any indeterminate times, such as the delays at the beginning and the end of the TFG, are given definite values just before or during run-time.

The OCPN model of the same scenario is as follows:

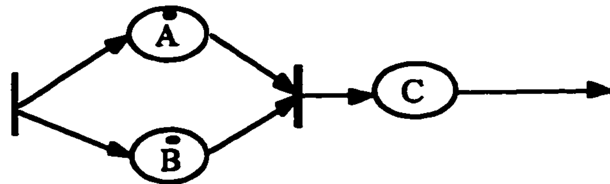

Figure 8: OCPN model of the scenario

As in the TFG, an ellipse or a circle represents an object with a duration. This is called a place in OCPN terminology. Transitions, represented by the vertical lines, activate and deactivate places. A transition fires, activating the places which follow it, when all of its input places has a token (represented by the dot). The transition moves the tokens from its input places to its output places instantaneously. A place is activated when it receives a token, and remains active for its specified duration, thus "locking,, the token. The place becomes inactive when it releases the token to the transition. The arcs (arrows) show in which direction the token progresses. The OCPN shown above can be expressed as "Objects A and B, both of same duration, start and end simultaneously. When they both end, object C is activated. Once object C becomes inactive, the scenario terminates.,, This is also equivalent to the scenario given above. Times in the OCPN formalism are usually hard times, because they represent the duration of multimedia objects, which are usually known beforehand.

The advantage of the TFG over the OCPN is that if one of the objects arrives late, the TFG doesn't have to be modified, whereas the OCPN must be modified. If, for instance, B arrives late, the first interval δ will be greater than zero, and A will be delayed until B is ready to be presented. Since the interval between the end of C and the end of the presentation is not determined either, B can be delayed by any amount, and the TFG

will still be valid - C will simply be delayed by the same amount that B was. The OCPN, on the other hand, doesn't have "waiting,, places between the transition which starts the scenario and the beginning of A and B, so if B is late, the OCPN is no longer valid. Of course, if the delay of B is predictable, say b seconds, then a waiting place of b seconds can be inserted into the scenario between the first transition and the beginning of A and B. The problem is that b is not predictable - at least not in a deterministic, precise way, which is necessary for the OCPN. Therefore, the scenario must be changed every time B, or any other object, is late. It should be noted that while a Petri net model of some event can accommodate delays and variations by using probabilistic times for places, the OCPN uses hard times, because it is not a model for a presentation, but rather a schedule for a presentation - it tells an application when to present objects, instead of telling someone studying the application when the object might have been presented. In brief, an OCPN specification requires that a presentation scenario be specified (i.e. duration of objects and waiting intervals be known) to a greater degree than does a TFG.

This is especially important for the correction of synchronization errors. Common sense also tells us that, if two objects are to be presented in parallel and one is late with relation to the other, it makes sense to delay the earlier object so that both can be presented together, as long as this delay is acceptable to the user and it doesn't disrupt the rest of the presentation. In other words, the relationships between the elements of the presentation are more important than the actual times at which each element is displayed. The flexibility of the TFG method ensures that a given scenario can generate many valid presentation schedules. The SSP adjusts the presentation scenario if objects arrive late; if the specification method wasn't flexible enough, the new schedule would probably violate

the original specification and disrupt the whole presentation. This doesn't happen with the TFG, which shows how much the schedule can be altered before the rest of the presentation is affected. The impact of the TFG on scheduling and the efficiency of error correction procedures is discussed in more detail in [Li94]. A discussion on the relative merits of Petri net based specifications can be found in [SN95], and also in [Li94].

The next step is generating presentation and delivery schedules. The algorithm for deriving a presentation schedule from a TFG is given in [LKG94.2]. There is a precise relationship between the presentation schedule and the delivery schedule. If an object is to be presented at time *P*, it must be sent by the server at time *P-D*, where *D* represents the delays that the object will incur before it is presented to the user. These delays are the end-to-end network delay on the connection, the time it takes to decode the data (if any), and the buffering time at the client, which is used to compensate for predictable network delays [LLG94] [LLBG96]. Thus, the delivery schedule depends on the presentation schedule. But which presentation schedule? As stated earlier, the same presentation can have many different presentation schedules, and it is likely that this schedule will change during the course of the presentation. But the presentation schedules for a given scenario are all related: they can all be expressed as schedules with no added delays (i.e. no synchronization errors) plus any delays incurred during presentation. This "no-delay" schedule is the one that serves as the basis for the delivery schedule. The delivery schedule is only computed once; any adjustments to the presentation schedule are in fact a matter of synchronization recovery and do not concern the initial scheduling of the presentation.

Thus, the relationship between the different temporal specification structures for a presentation is as follows: the TFG is generated from the scenario, the presentation schedule is generated from the TFG, and the delivery schedule is generated from the presentation schedule. The client only needs to generate the presentation schedule, while the servers must each go through all the steps.

4.1.3 Functionalites provided by the distributed scheduler

It is clear from the previous section that the distributed scheduler provides both synchronization specification and object-layer scheduling, both of which are lacking from the centralized scheduler. These are essential components of an effective synchronization strategy. These algorithms could all be executed centrally - e.g. at the client - but since the delivery schedule involves the servers, it makes more sense to relieve the client of this responsibility, thus making the client application simpler and smaller.

The synchronization functionalities of the distributed scheduler system are summarized in figure 9.

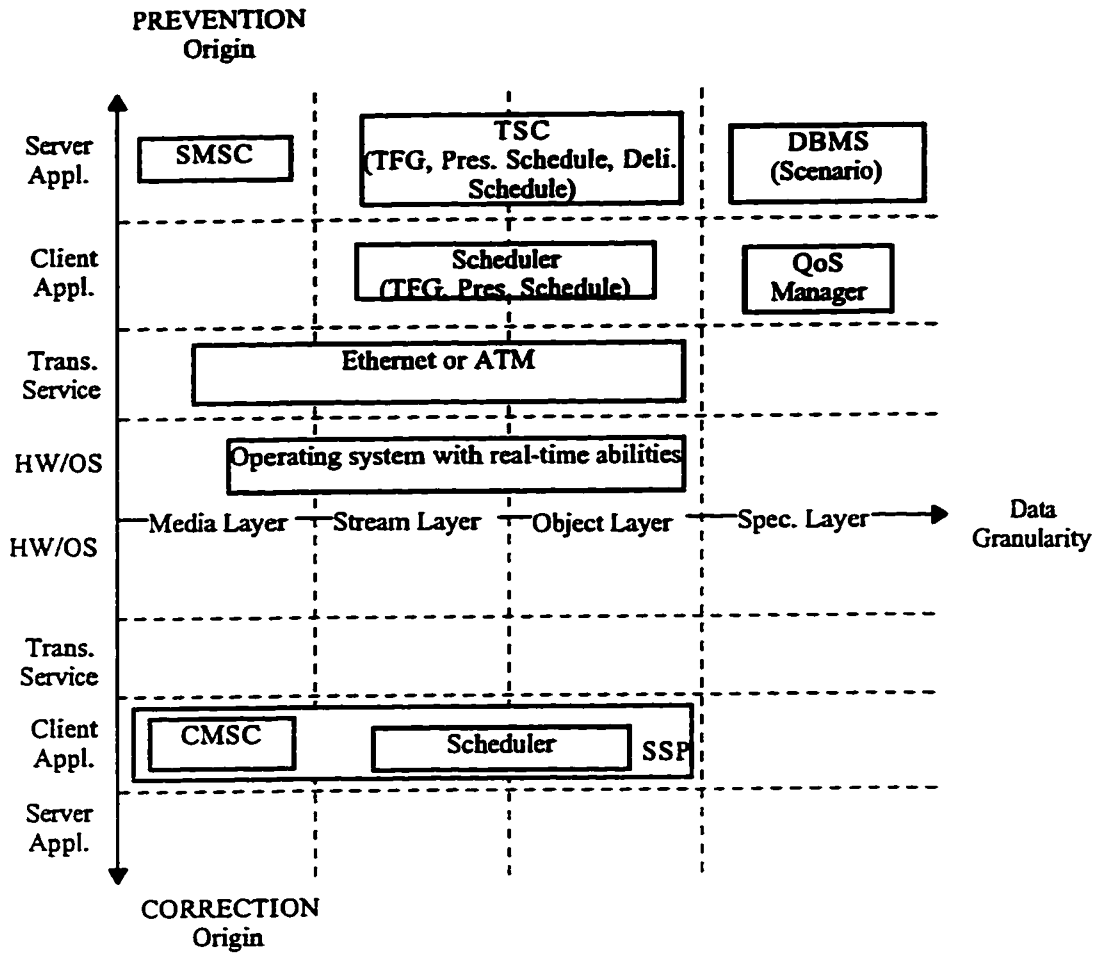

Figure 9: Analysis of the Distributed Scheduler System Architecture

4.2 Moving from a Centralized to a Distributed scheduler

4.2.1 Similarities between both models

So far, we have seen that the distributed model provides more comprehensive and theoretically more efficient scheduling than the centralized model. But does this mean that the current centralized prototype must be entirely re-designed?

To answer this question, we must first see how much of the current prototype is still useful. As we have seen earlier, the synchronization recovery algorithms and system components are essentially the same in both models, and thus remain the same in the distributed model. Furthermore, the discussion of chapter 3 indicates that the CMFS fulfills most of the roles of the SMSC in the distributed model: it controls the server-end of the connection and provides media-layer scheduling, which are responsibilities of the SMSC. It is also able to determine the network delay on a connection with the prepareBound parameter in the `CmfsOpen` interface, which allows a client to open a connection [MHN96]. The main difference between the CMFS and the SMSC is that the SMSC initiates the connection, while the CMFS waits for a message from the client before opening up a connection. This is not desirable in a distributed scheduler architecture, because the server must be able to initiate connections to obtain the network delay for the delivery schedule. However, there should be a way of making use of the functionalities already provided by the CMFS in the distributed model.

The client scheduler in the centralized model is very rudimentary and doesn't handle TFGs; it is also deficient as far as object- or stream-layer scheduling goes. However, a scheduler on the client side is still necessary in a distributed scheduler system, and thus this element should be included in the new architecture as well.

4.2.2 What needs to be changed in the current system

Clearly, TSCs must be added to the servers. However, since the CMFS basically fulfills the role of the SMSC, the TSCs should utilize the CMFS, and not be a part of the CMFS. The CMFS must be changed so that the TSCs can tell it to open a connection to the client; the TSCs will then be able to obtain the network delay, derive a delivery schedule, and deliver data to the client at the most suitable times. The TSC will in fact coordinate the functioning of the server, by telling the CMFS what to do. The scheduling algorithms at the client side must take into effect the stream- and object-layers.

The changes to the CMFS and the client scheduler simply involve modifying existing code. The TSCs, however, need to be written from scratch. They must be able to communicate with the client, communicate with the CMFS, and derive a delivery schedule. The following chapter discusses how the new scheduler system is designed and implemented.

Table 1 shows the correspondence between the components of the system with the centralized scheduler and the system with the distributed scheduler.

Component of the Distributed Model	Corresponding Component in the Centralized model
Client scheduler	Scheduler
CMSC	MSC
GUI	GUI
Database server	Database server
Audio/Video server	CMFS
Text & Image server	UofA server
SMSC	CMFS or UofA server
TSC	None (the client scheduler fulfills some of the functions of the TSC).
QoS Manager	QoS Manager

Table 1: Correspondence between the components of both systems

CHAPTER 5: IMPLEMENTING THE DISTRIBUTED SCHEDULER

5.1 Designing the Architecture

5.1.1 Adapting the distributed scheduler to the News-on-Demand system

The distributed scheduler architecture given in the previous chapter is quite sophisticated. Given that this scheduler has to be implemented in such a way that the effect on the other components is minimal, it is to be expected that the theoretical architecture will have to be adapted to the current system.

First of all, as many as possible of the components from the centralized model have to be used in the distributed model. Table 1 shows that all of the former's components have a place in the architecture of the latter. This means that the task consists mainly of adding the TSC modules to the servers; they now consist of both a TSC and a SMSC, the latter being either in the CMFS or the University of Alberta server, as the case may be. Since the TSCs use these servers, much as a client would, as opposed to being integrated into them, these TSCs look like clients to the media servers. However, since the client does not do any of the delivery scheduling any more, it now communicates with the TSCs, not the media servers. The TSCs look like servers to the

client. The TSC is effectively a proxy server between the client and the media server, allowing the client to request many different objects from the same server with just one command. While the overall architecture still fits the client-server model, the servers themselves now have a two-tiered architecture, with the TSC handling incoming control messages and sending commands to the media servers, while the media servers control the data connections. This is illustrated in figure 6.

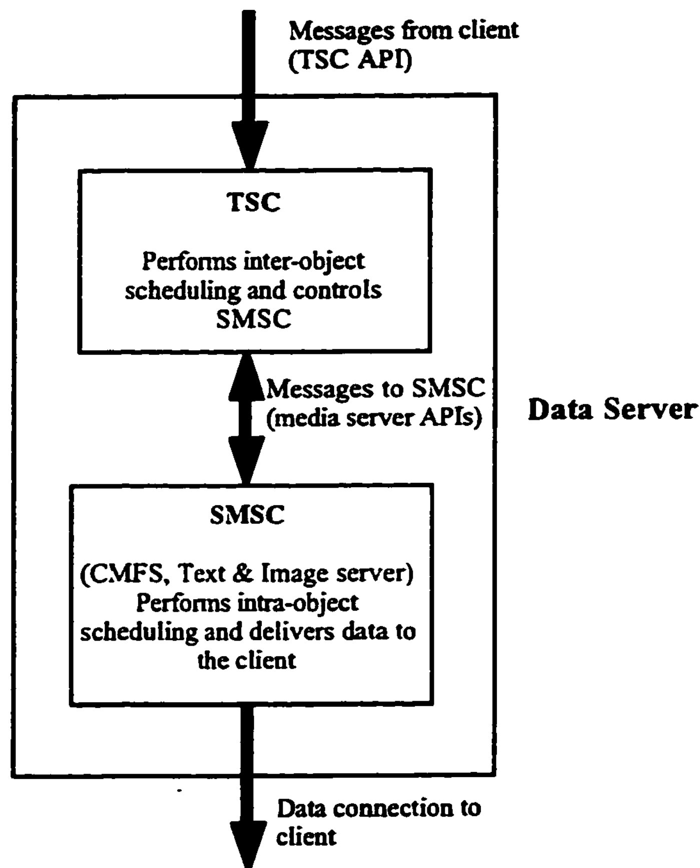

Figure 10: Two-tiered server architecture

Of course, this does not mean that any of the other elements of the system don't need to be changed either. Most obviously, the scheduler algorithm at the client must be more complete, in the sense that it must now be able to schedule streams and objects, not

just perform intra-stream scheduling. Furthermore, the interfaces between the client and the servers must be changed. In fact, two types of client interfaces are now needed: an interface to the TSCs for server control, and interfaces for the data connections, which are simplified versions of the original server interfaces.

The TSCs are now in charge of inter-object delivery scheduling, which means that they are responsible for setting up data connections to the client and starting the delivery of data at the right times. Since the TSCs communicate with the media servers as a client would, i.e. using the server APIs, these APIs must allow the TSCs to open connections to the client and schedule data retrieval. This is not the case with the server APIs in the centralized model. New APIs are needed to make the TSCs feasible.

The design of the CMFS introduces another constraint: the CMFS API is designed in such a way that a client can only connect to one CMFS at a time. Theoretical models of the News-on-Demand system usually have many separate media servers, and the distributed architecture from chapter 4 is no exception. The CMFS's architecture does not in fact violate this requirement, because one logical CMFS server can, and in fact should be, implemented on many different machines[MHN96]. Thus, even though the client only sees one CMFS, it is actually accessing many servers. As far as the design of the distributed scheduler goes, though, there is one (logical) CMFS in the system, which handles all of the time-dependent media for the system, and thus only one TSC for these media.

An implementable architecture for the system with the distributed scheduler can now be given.

5.1.2 The architecture of the new system

Figure 7 depicts the new architecture.

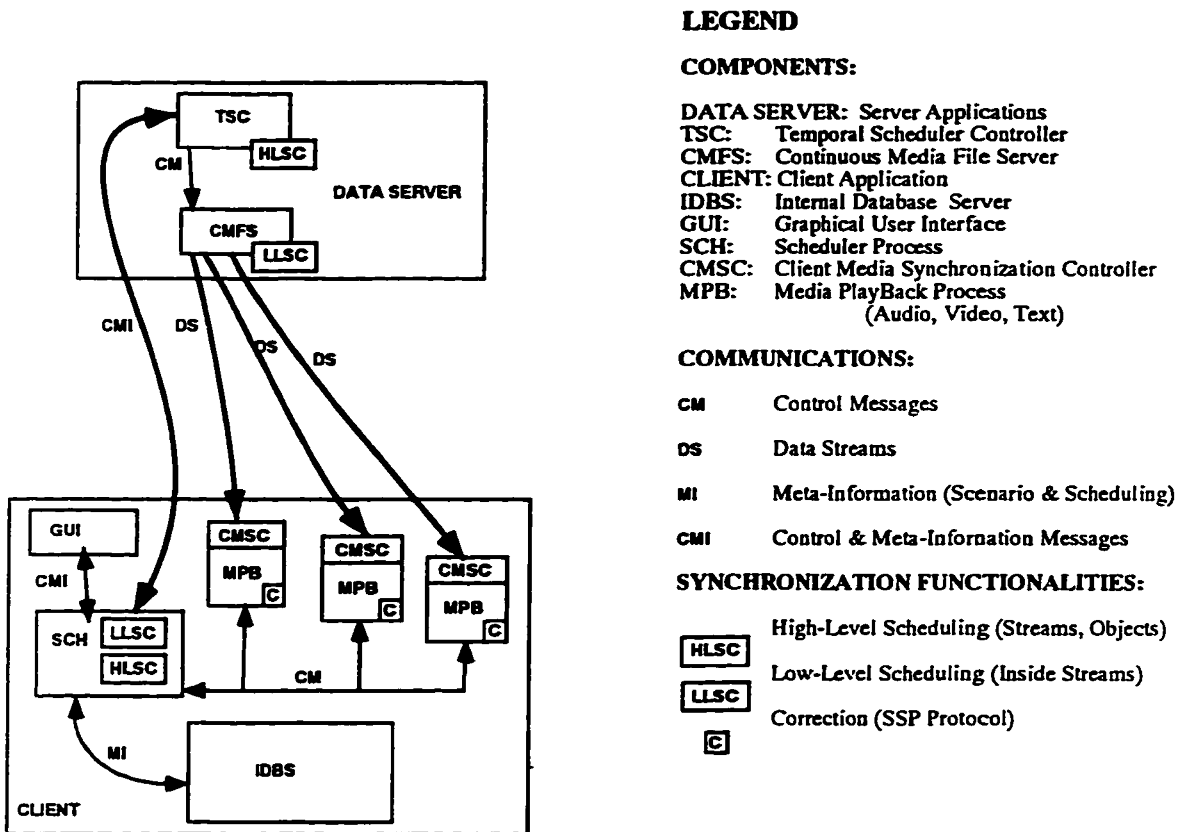

Figure 11: Implemented architecture of the distributed scheduler

The architecture has visibly undergone some changes during implementation. The most obvious difference with the theoretical architecture is that there is only one time-dependent media server, no time-independent media servers, and the database server is

now in fact a local database on the client. This is due to the fact that the APIs for these two servers were not available during implementation. It was thus decided to store all of the meta-information on the client, while text captions for the articles were treated like data streams and stored on the CMFS, because they have to be synchronized with the audio streams. The GUI module on the client also simulates the QoS managers by providing QoS parameters, also stored at the client, to the scheduler. These simplifications allow the scheduler system to be tested without worrying about how to integrate all of the components into one system, and thus do not compromise the validity of this research. However, the issue of TSCs for the text and image server will have to be addressed in the future.

As far as the operations of the scheduler go, it should be noted that the TSCs do not communicate to compute a delivery schedule jointly any more. This joint coordination was supposed to ensure that the different "sub-schedules„ which are supposed to be derived on each server are all consistent. However, coordinating the servers is a complicated task. The main problem is that all the servers need some kind of common time reference, be it a start signal broadcast to all the servers or a global clock. Also, in the early stages of design, when the multiple CMFS model was still being considered, making all of the servers aware of each other appeared to be a relatively complicated matter. At any rate, since there is only one continuous media server, this lack of coordination should not be a problem.

Another simplification to the theoretical architecture is that the TSC only computes the delivery schedule, based on the presentation schedule derived by the client. The theoretical model from chapter 4 does not address this explicitly. In an architecture where there are many data servers, the schedules must be the same on all the servers to ensure consistency, so any approach requiring each TSC to do all of the computations which lead from the scenario to the delivery schedule involves a lot of redundancy. This is not currently a problem, as only one data server is being used, but could become one when the text and image server gets integrated into the system. Furthermore, the client would also be deriving the same TFG and presentation schedule as the server in these strategies. The solution that was adopted was that the client would first send the scenario to the TSC, which would know how many connections need to be opened and which object are involved in the presentation. Then, when the presentation schedule is derived at the client, it is sent to the TSC which then derives the delivery schedule. So the TSC only derives the delivery schedule, while the client derives everything else.

Obviously, in this architecture the APIs for the media servers incorporate the changes which were necessary for the proper functioning of the TSCs. The client has also been changed: its scheduler can handle high-level scheduling, and it communicates with the media servers via the TSCs. As a result, the CMSCs do not open connections and request objects any more.

The following sections will examine how all of these changes were made.

5.2 Changes to the CMFS

The CMFS API implemented for the centralized scheduler does not allow a process other than the client to open connections. This is a problem for the distributed scheduler because the servers need to initiate the connections in this model, so that they know the network delay on each connection; furthermore, streams and connections are bound together in the CMFS API, so the servers also need to open the connections to be able to control the streams at all. The CMFS design team agreed to change the `CmfsOpen` interface, which opens connections and associates them to a stream. The new interface, `CmfsProxyOpen`, allows a third process to tell the CMFS to open connections to the client, as long as the third process has enough information about the client, and returns two parameters: `prepareBound`, which gives an upper bound on the time required for a call to the `CmfsPrepare` interface to complete, and a connection identifier [MHN96].

`CmfsPrepare` is an interface which allows a client or a third process, such as the TSC, to request that a stream be readied for delivery [MHN96]. It effectively allows the calling process to initiate the delivery of data. `CmfsPrepare` does not allow a stream to be scheduled into the future - delivery begins almost as soon as the `CmfsPrepare` call is received by the CMFS. This is a disadvantage for the TSC, because the TSC should be able to inform the client whether the preparation of the streams succeeded, and if the TSC has to wait before making the call to `CmfsPrepare`, it will generally not be able to do so. The solution proposed by the CMFS team is to modify `CmfsPrepare` so that it can

take a parameter indicating the time at which delivery is to commence, so that the CMFS reports on the status of the stream before delivery starts. However, at the time of writing, this had not yet been implemented.

The CMFS needs to keep track of the TSC which is associated to it. This is necessary because the client application is only aware of the CMFS, not the TSC. It was stated earlier that the CMFS has a distributed design, but that the client only sees one server. This is because the client is given only the address[2] of the machine running the CMFS administrator process, not the addresses of the individual servers [MHN96]. The client end of the CMFS API is given the address of the sub-servers, but they are unknown to the client application as such. To make the new design consistent with the old one, and to ensure that each logical server is known by only one address, the CMFS needs to be informed of the location of the TSC associated to so it can pass on this information to the client TSC API. This requires some changes so that the CMFS can accept a message from the TSC and register it. The way in which this should be done has not been officially agreed upon yet, but a temporary solution allows the TSC to register with the CMFS in such a way that it looks like a sub-server to the CMFS.

---

[2] The RTT environment uses identifiers called RttThreadIds to locate processes (called threads) and to allow communications between them. The RttThread is a structure which stores the IP address of the machine running the thread, the port number of the UNIX process running the thread, and a number unique to the thread.

5.3 New Data Structures Used in the Scheduler

This section discusses the important aspects of the data structures. A comprehensive list of data structures is given in the appendix.

The data structures needed for the distributed scheduler are: a scenario, a presentation schedule, a delivery schedule, and control messages to allow the client and the TSC to communicate. The approach taken with the scenario and the schedules makes use of the fact that a presentation schedule can be thought of as a scenario with times added in. Because the exact structure of the scenario hasn't been officially agreed upon yet by the CITR teams, a temporary scenario structure was designed for the client scheduler. This was presented in section 3.3.2. This is a hierarchical structure, with a scenario being made up of the list of the streams that make up the presentation, each stream being a list of LDUs, called segments. The transition from scenario to schedule is made by filling in the start times of the streams (this is a new addition for the distributed scheduler) at the client scheduler. The delivery and the presentation schedules are the same type of structure; the main difference is in the value of the start times.

However, the delivery schedule is only needed at the TSC, which only does stream and object layer scheduling; the segments structures are unnecessary, as are other information pertaining to the client scheduler, such as the "epoch,„ which is the absolute start time of the presentation. Thus there is a version of the scenario/schedule for the client, as seen above, and a simplified version for the TSC. `Document`, the structure that is passed to and used by the TSC, contains identifiers for the objects in the presentation,

their delivery times, their connections, their media type, and so on. The presentation times are provided by the client when the user requests the start of the presentation. The algorithms for deriving these structures are given in a later section. Note that the TFG structure is not used here, for practical reasons; as was already discussed, the scenarios used in this system are quite simple, and can be specified and modeled using simple means. Nonetheless, future work should include the TFGs in the scheduler algorithms.

The control messages allow the client to send requests and meta-information to the TSC at the same time. This meta-information can be the `Document` structure, the presentation parameters for each stream (necessary for `CmfsPrepare` [MHN96]), or simply an identifier for a given document. The algorithms which use theses structures, and the commands which accompany them, are explained in a later section of this chapter.

5.4 Implementation of the TSC

5.4.1 General requirements

As was previously mentioned, the TSCs look like servers to the client and like clients to the media servers. This means that the TSCs need to receive messages from the client and communicate with the servers using their APIs. At the time of implementation, the API for the University of Alberta server was unavailable, so only a TSC for the CMFS was implemented.

The functionalities that the CMFS provides for the TSC are the ability to open connections to the client, the ability to ready streams for delivery, the ability to stop delivery of streams and the ability to close connections. The functionalities that must be provided by the TSCs are: deriving a delivery schedule from the presentation schedule, coordinating the opening of all of the connections needed for the presentation, readying the whole article for delivery, stopping the article and closing the connections. The last four operations are carried out by using the CMFS API. The TSC must also report to the client on the status of its operations.

5.4.2 Functioning of the TSC

The functioning of the TSC module can be represented by the following state diagram:

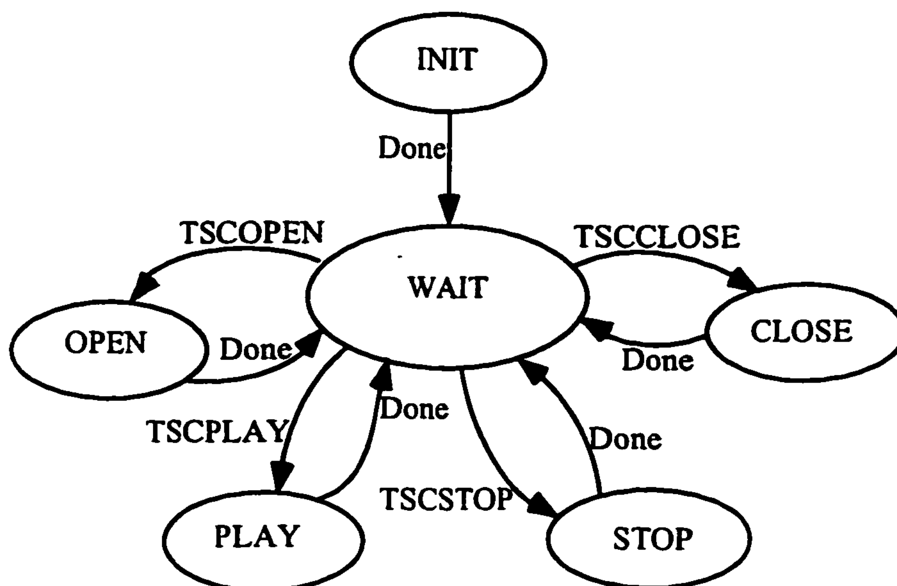

Figure 12: State diagram of the TSC

The states in the diagram are as follows: the INIT state is the state where the TSC initializes its data structures and network environment and registers with the CMFS. The WAIT state is the state in which the TSC waits for messages from the client. In the OPEN state, the TSC uses the scenario received from the client to open all of the necessary connections between the CMFS and the client. In the PLAY state, the TSC uses the presentation schedule received from the client to derive a delivery schedule and to make the CMFS start sending data streams according to the delivery schedule. In the STOP state, the TSC instructs the CMFS to stop sending data. In the CLOSE state, the TSC tears down the server-end of the data connections. All of these states are entered to when the appropriate message (TSCOPEN, TSCPLAY, TSCSTOP, TSCCLOSE) is received from the client. When the operations associated to these messages are carried out, the TSC goes back to the WAIT state, and the status of the operations is reported back to the client.

The algorithm is straightforward. After a few initialization steps, the TSC simply waits for messages from the client and acts based on the type of message received. The step "Register with CMFS„ makes the TSC known to the CMFS involved in the presentation, as was discussed in section 5.2.

The TSC is designed to handle many different articles at once, up to a maximum of five. This is because the same server can provide data for many clients. The number five is in fact arbitrary, but it must be kept in mind that the CMFS can handle a maximum number of connections, and that no new articles can be presented if the CMFS is

saturated (this value is currently set to 15, but this can also be changed). The scenario for each article that has been requested from a given TSC/CMFS combination is remembered by the TSC, and each scenario is given an identifier. Once the scenario for an article is known, all subsequent operations on the article can refer to the identifier instead of giving the full scenario.

The algorithms for opening and closing the connections, and stopping the delivery of the streams, are very simple - they just involve going through the scenario, finding the individual streams or connections, and carrying out the required calls to the CMFS. The algorithm for deriving the delivery schedule is more complicated, and it is explained in the next section.

5.4.3 Delivery schedule algorithm

The delivery schedule algorithm is basically the same as the one presented in section 4.1.2: the delivery time is the delay encountered by the stream before it is presented to the user subtracted from the desired presentation time. To simplify things, the delay on the stream was taken to be the network delay only, since the decoder and buffering delays at the client are assumed to be negligible. The network delay is provided by the prepareBound parameter, seen earlier. Its value is the network delay plus three "time slots,,. A time slot is the internal timing unit of the CMFS, and lasts about half a second [NMH96]. Thus this parameter also takes into account the time it takes the

CMFS to process the request, and thus is a better parameter than the network delay alone.

Another thing to take into account is that the MCRLab synchronization algorithms are designed to operate without needing to synchronize the clocks of the machines running the synchronization system. This means that the delivery schedule cannot in fact be based on the absolute times in the presentation schedule. So the presentation schedule that the TSC receives is a schedule with relative times. Each stream has a start time that is an offset from the start time of the first stream in the presentation. Naturally, the first stream has a start time of zero. The delivery time is therefore also in relative times. These times are the times in the presentation schedule adjusted for the different network delays that are encountered on the different connections. The absolute times are determined by the time at which the "Play„ command is received at the TSC, which becomes the absolute starting time for the delivery schedule.

The algorithm for deriving the delivery schedule is:

```
d_i = p_i - pb_i
offset = abs(min(d_i))
d_i = d_i + offset
```

for every stream in the scenario, where d_i is the delivery time of stream i, p_i is the presentation time of stream i, pb_i is the prepareBound (i.e. delay) parameter for stream i, and `offset` is the absolute value of the earliest delivery time. Since the smallest value for p_i is 0, the smallest value for d_i will be negative, which is unacceptable because this implies that the TSC has to schedule objects in the past. Instead, the value of

the earliest delivery time is used as an offset to make all the delivery times positive, with the earliest delivery time now being zero.

Since the `CmfsPrepare` interface still does not allow streams to be scheduled into the future, as discussed in section 5.2, the TSC must schedule the times at which it calls `CmfsPrepare`. To ensure that the streams are prepared in parallel so that the delivery times are respected, the TSC makes use of the RTT environment to create parallel threads [FHM+95] which execute the calls to `CmfsPrepare` and which are activated according to the delivery schedule. The TSC replies to the client before all the streams are prepared, otherwise the client would be forced to wait until the last object was prepared before continuing with the presentation. This goes against the requirement that the TSC should report the status of its operations to the client, future researchers should address it as soon as the necessary change to `CmfsPrepare` is made.

5.5 Implementation of the TSC API

The TSC API provides the client application with a number of interfaces which allow the client to communicate with the TSC. Each interface is presented and explained; then, the implementation of the API itself is discussed.

5.5.1 TscOpen

The TscOpen interface allows the client to send the scenario to the TSCs and request the opening of the necessary data connections between the client and the media servers. Its format is

```
TscStatus TscOpen(Scenario *scenario);
```

TscStatus is an enumerated C data type used for status codes by TSC. The parameter *scenario is a pointer to a structure of type Scenario which was discussed earlier.

TscOpen takes a pointer to a scenario structure and maps the contents of this structure into the Document structure which is understood by the TSC. It then (in theory) determines how many servers are involved in the presentation and sends a message of type TSCOPEN, which contains a copy of the Document structure, to each. In actual fact, since the API for the text and image server is not available, the message is only sent to the TSC associated with the CMFS. If the TSC returns a message indicating successful completion of its operations, TscOpen returns TSCOK as a status code. If not, the status code returned by the TSC is returned to the client. Assuming the operations at the TSC were successful, the TSC returns an article identifier to the API, which is hidden from the client. This identifier is used in all subsequent communications between the client and the TSC for the duration of the presentation.

5.5.2 `TscPlay`

The `TscPlay` interface allows the client to request the commencement of the presentation. Its format is

```
TscStatus TscPlay(PlayParms *parameters);
```

`PlayParms` is a structure composed of the parameters needed to execute the `CmfsPrepare` calls at the TSC. These parameters, explained in [MHN96], determine the start and stop positions, the playback speed, and the skip values for each stream. `Skip` is a parameter which determines how many LDUs (referred to as sequences in the CMFS documentation) in the stream are to be skipped for every LDU sent. There is also another parameter which gives the presentation time of the stream, from which the delivery time is derived. Each stream has a set of these parameters associated to it in the structure.

The `CmfsPrepare` parameters are sent to the TSC in a message of type `TSCPLAY`. The status code returned by the TSC is returned to the client. As was noted before, the TSC does not actually return the status of the `CmfsPrepare` operations, because this would require the TSC to make the client wait unnecessarily. The client is therefore only informed of communications problems between the TSC and the client, such as a failure of the TSC to receive the message.

The `TscPlay` interface allows the client to request fast-forward or rewind operations. The `CmfsPrepare` parameters can be specified in such a way that a stream can be presented forwards or backwards and at different speeds. Thus any variations to the speed or direction of the presentation simply require the appropriate parameters to be given to the `TscPlay` interface.

The `TscPlay` interface also accepts a null pointer as a parameter. In this case, the parameters for each stream are set to default values - the start position for each stream is the beginning of the stream, the end position is the end of the stream, the speed is the original speed of the stream, the skip parameter is set to zero, and all streams start at the same time. This option was provided because this type of scenario, with all streams running from start to finish in parallel, appears to be the most common in the current trials of the News-on-Demand prototype.

It should be noted that this interface will have to take into account the text and image server when the API for it becomes available.

5.5.3 TscStop and TscClose

`TscStop` allows the client to stop the presentation. `TscClose` informs the TSCs that the client wishes to quit the presentation. They both have similar formats:

```
TscStatus TscStop();

TscStatus TscClose();
```

These interfaces take no parameters. The API has been given an article identifier by the TSC, so it doesn't need any additional information from the client to execute them. They simply send a message of type `TSCSTOP` or `TSCCLOSE`, respectively, to the TSCs involved in the presentation. The status of the processing of these messages by the TSCs is returned to the client.

A possible improvement to the `TscStop` interface might be allowing specific streams to be designated for stopping; however, the usefulness of this option must first be ascertained. `TscClose`, however, is used to close the entire presentation, and thus it makes sense that no parameters are required for it.

The `TscStop` interface should be used with caution. The `CmfsStop` interface called by the TSC upon receiving the `TSCSTOP` message doesn't seem to work properly, and the TSC returns a status code of `TSCUNABLE`, indicating failure to carry out the `CmfsStop` calls. The call to `CmfsStop` often makes either the CMFS or the client crash as well. However, it appears that it is enough to stop the streams at the client and to close the connections with `TscClose`, if the user wishes to stop viewing an article.

5.5.4 API implementation

The API consists of the interfaces just described and a number of data structures, seen earlier, which are stored on he client machine but are hidden from the client application. The article identifier mentioned earlier is such a data structure. Other data structures are the addresses of the TSCs involved in the presentation, and the messages that the API uses to communicate with the TSCs. The API must first determine the addresses of the TSCs by querying the media servers which are associated to each object in the scenario. As of this writing, this is done by having the client end of the API query the CMFS.

Each interface follows essentially the same algorithm: the parameters passed by the client, if any, are formatted for the TSCs, the messages for the TSCs are sent, and the replies of the TSCs are interpreted and passed on to the client. In addition, `TscOpen` must find the address of the TSC before proceeding; differs in that it must find the address of the TSC first; `TscClose` must discard the information about the presentation - article identifier, TSC locations - which was established by `TscOpen`.

5.6 Integration with the Overall Prototype.

Integration with the overall prototype mainly requires changes to the server and client applications. The changes to the servers were discussed in section 5.2. The changes

at the client affect mainly the architecture of the scheduler, which was implemented by MCRLab researchers [BG96]. The Scenario data structure previously discussed has been changed, as shown in section 5.3. Since this structure now contains the identifiers (UOIs) for each stream, the scheduler was changed so that it obtains this information from the (internal) database server and stores it in the scenario (previously, each data reader process would obtain the UOI for its stream, and this information was unknown to the scheduler or in the overall scenario). The scheduler must also perform the presentation scheduling at the inter-stream level, not just the intra-stream level, although, as discussed earlier, this scheduling is trivial for the moment, as the typical scenario of an article is extremely simple. Also, whereas the scheduler did not communicate with the servers in the centralized version, leaving the management of the data streams to the data reader processes, the client scheduler communicates directly with the TSC in the distributed version, leaving the data readers in charge only of listening for connections from the servers and reading data. The modified client application performs only minimal maintenance on data streams: the data readers listen for their connections, start reading data according to the presentation schedule, and stop and close the data streams when the presentation is over or when the user desires to stop it. Thus, the client now uses only four interfaces from the CMFS API[MHN96]: CmfsListen, CmfsRead, CmfsStop and CmfsClose. The data readers do not need to call CmfsPrepare or determine the appropriate parameters for this interface anymore, so their code is in fact simplified. The client now uses two APIs. The first one is the TSC API, which is used for control messages between the scheduler and the servers. Given the problems with TscStop, only TscOpen, TscPlay and TscClose are currently used by the client scheduler.

The second is the simplified CMFS API, as described above, which is used by the data readers.

The distributed scheduler software has been successfully integrated into the overall prototype at the MCRLab. No formal performance testing has been carried out, and will be carried out by other researchers. As far as the user is concerned, the changes to the system should be invisible; everything should work exactly as before. The only difference should be in the performance of the synchronization algorithms. Future work on performance testing should give a quantitative appraisal of the performance gains and of the other effects of the inclusion of a distributed scheduler into the News-on-Demand system.

5.7 Functionalities of the Distributed Scheduler System

The functionalities of the new system are given in figure 13.

We can see that the functionalities are essentially the same as in those seen in the original distributed scheduler design. Some of the components have different locations, the client scheduler is involved in media layer scheduling, there is no Text and Image server, and the data structures aren't the same as in figure 9, but scheduling at all layers of data granularity is addressed. Also, we see that the client is also involved in specification layer correction operations when the GUI, acting as the QoS manager, simulates a quality of

service renegotiation. The comparison of figures 9 and 13 also shows how the components in the implemented system correspond to those in the original design for the distributed scheduler.

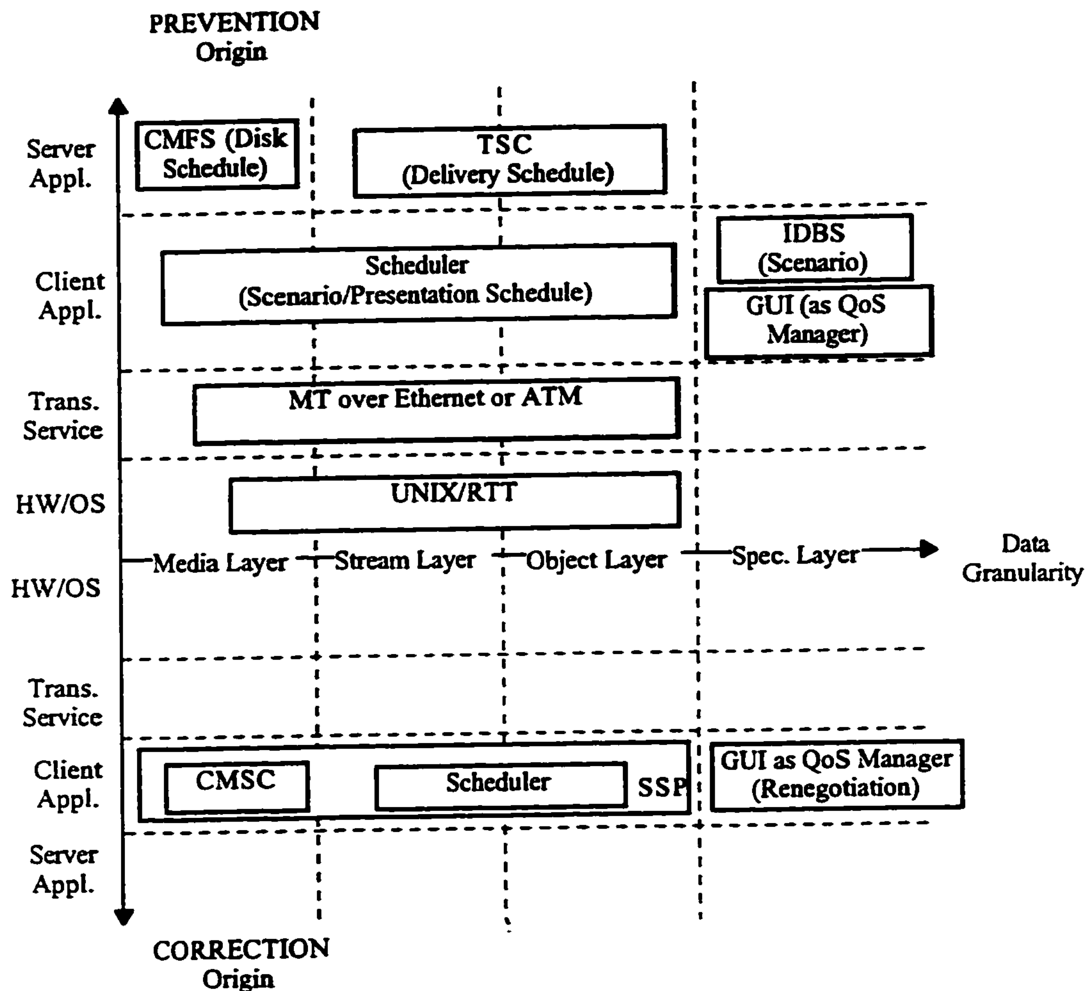

Figure 13: Analysis of the Implemented Distributed Scheduler System

CHAPTER 6: CONCLUSION

6.1 *Summary of Thesis*

This summary discusses the design and implementation of a distributed scheduler for the CITR News-on-Demand system prototype. A theoretical framework for doing this was given. This framework allows the synchronization aspects of a multimedia system to be qualitatively studied along three axes: the types of synchronization phenomena occurring in the system as they relate to the granularity of the data; the components of the system where these phenomena occur; the type of strategy that is used for dealing with synchronization errors. The centralized scheduler that was originally part of the system was analyzed with this framework. It was found to be deficient in the area of prevention operations, namely inter-object and inter-stream delivery scheduling, both at the client and at the servers. However, it does provide intra-stream scheduling at the servers and at the client, and both inter- and intra-stream error recovery are supported. The distributed scheduler algorithm developed at the MCRLab was presented and explained; it provides for both prevention and recovery operations at all levels of media granularity, in the form of the scheduling algorithms which are executed jointly by the client and the servers and the Stream Synchronization protocol executed at the client, respectively. Thus it was shown that this architecture can make up for the shortcomings of the centralized scheduler. The adaptation of the distributed scheduler into the News-on-Demand system was discussed. It was shown that many of the components of the

new scheduler were already present in the old scheduler, and that what was needed was mainly adding scheduling processes at the servers. These processes, known as TSCs, are designed to use the media servers in use in the centralized scheduler system, so the architecture of the servers becomes two-tiered. Other changes to the centralized architecture include more complete scheduling algorithms and data structures for the client and more flexible APIs for the media servers. It was found that certain aspects of the original architecture for the distributed scheduler had to be simplified, namely the TSCs need only derive a delivery schedule, based on the presentation schedule provided by the client, and each server derives its schedule independently of the others. The number of servers in the system is also smaller than what was foreseen in the original architecture, which reduced the need for interaction between the servers. On the whole, however, a distributed scheduler system, with scheduler processes on the client and on the servers, using most of the MCRLab's scheduling algorithms, was successfully implemented and integrated into the overall system..

6.2 Contributions of this Thesis Work

6.2.1 Implementation of a distributed scheduler

A distributed scheduler that is compatible with the rest of the News-on-Demand system has been successfully implemented. This scheduler solves the problem of insufficient scheduling functionalities in the centralized scheduler system and makes use

of the synchronization algorithms of the MCRLab. The scheduler has been integrated into the MCRLab's News-on-Demand prototype; basically, all that remains is the fine-tuning of certain parameters in the TSC module [Jar97].

6.2.2 A conceptual framework for studying synchronization systems

This framework builds on the synchronization taxonomy work of Steinmetz, Meyer, Effelsberg and Blakowski, which can be found in [SN95] [MES94] [BS94]. This work provides a way of classifying synchronization phenomena and systems according to the level of media granularity that is addressed. However, this model does not distinguish between the kinds of synchronization operations being carried out on the data (i.e. prevention and recovery), and between the parts of the system which actually perform the operations. These aspects are important because they give qualitative insight into the performance of a system. A system that performs only recovery operations on data will probably spend more time than necessary on correcting synchronization errors, thus compromising performance; similarly, a system with only prevention operations is sure to have poor synchronization performance at the client. A system which centralizes all of its synchronization operations, particularly scheduling, at the client will make heavy demands on the resources of the client, potentially compromising performance as well. Thus, we can see that this three-axis framework allows to analyze and compare the synchronization performance of multimedia systems in a qualitative sense, which was not possible with the synchronization reference model in [SN95] alone. This is also useful

when designing a system, as certain qualitative assessments can be made without implementing it.

6.2.3 Introduction of a new server architecture

The analysis of the News-on-Demand system with the three-axis framework has highlighted the fact that servers are often designed to provide intra-stream or intra-object scheduling only. This is usually done so that the server is as simple as possible and suited to a wide range of applications. However, this places all of the responsibility of organizing the overall presentation and scheduling the delivery of objects to the client on some other component of the system, usually the client. This is not always desirable, as the client must also be kept simple to be suitable for many different hardware and software configurations, and might lead to inefficiencies in the scheduling algorithms, which have to derive a schedule for operations which are to be performed on another machine.

The server architecture proposed in this thesis allows the original design and functionalities of the servers to remain intact while also providing them with some scheduling abilities. The two-tiered architecture shown in figure 6 shows how this is done: a data server is composed of a TSC, which performs scheduling operations and handles all control interactions between the client and the media server, and the media server itself, which makes its data delivery and stream control functionalities available to the TSC, much in the way that they would be made available to a client application with an API.

The TSC does not in fact prevent the client from communicating directly with the media server; this is determined by the design of the application. However, this cooperation between a TSC, a dedicated scheduling process, and a media server allows simple and relatively low-level server designs (with regards to the level of media granularity they address) to be used in systems which require a distributed scheduler.

6.3 Future Work

While the main objective of this research, integrating a distributed scheduler into the News-on-Demand system, has been achieved, a few issues are still outstanding and should be looked into by future researchers.

6.3.1 Performance testing

Performance testing must be carried out to compare the relative performances of the centralized and the distributed schedulers. The purpose of this is to quantify the performance gains, if any, of the distributed scheduler. The aspects which should be studied are the performance of recovery operations at the client, the time it takes to initiate the presentation, and client and server CPU usage. The distributed scheduler should make recovery operations more efficient, as it should reduce synchronization errors in the data received by the client. It should also improve performance and reduce CPU usage by the client during the preparation phase, when the client is initializing its environment and performing scheduling algorithms, because the delivery scheduling and

the `CmfsPrepare` operations which tell the CMFS to prepare a stream for delivery are performed at the server. The CPU usage and time taken to prepare the presentation at the servers should increase because of the delivery scheduling algorithms, and it must be verified that this does not have a negative effect on the servers. More details on performance testing can be found in [Jar97].

This performance testing should allow certain parameters of the scheduler to be fine-tuned, such as the value used for the network delay as explained in section 5.4.3.

While no formal performance testing has been carried out yet, informal observation of the system seems to indicate that the client application crashes less often and loses slightly less video data with the distributed scheduler than with the centralized one; however, the client appears to need to re-synchronize the data streams more often with the distributed system [Jar97]. This would suggest that the load on the client CPU is significantly decreased in the new system, confirming the initial expectations for the distributed scheduler. The apparently reduced synchronization recovery performance might be due to improper setting the scheduler parameters mentioned earlier, or to an increased load on the server CPU. Formal performance testing should indicate whether these observations are valid and suggest whether changes need to be made to the distributed scheduler.

6.3.2 Improving the handling of many clients by the TSC

The TSC, like the media, server it is associated with, is designed to handle many clients at once. This is achieved by assigning each presentation a unique identifier at the TSC. Simple testing has shown that two clients can access the same server this way, as long as they are on separate machines. The CmfsProxyOpen interface used by the TSC determines the port for the connection it is opening based on a base value of 6780, the media type of the stream being opened and the relative position of the stream in the scenario. Thus, two clients on the same machine would be requesting data connections on the same port, which is not permitted by the system. This could be avoided by having the client chose the base value for the port and pass this value to the TSC.

Another issue is the fact that the TSC has been implemented as a single thread waiting for messages from the client. This means that the TSC cannot process new messages while it is opening connections or preparing streams. The TSC should be re-designed so that these operations are performed by concurrent threads which do not block the main thread of the TSC.

6.3.3 Implementing TSCs for systems with more than one server

This in fact covers two issues: implementing a TSC for the text and image server, and taking into account a possible multiple-CMFS architecture.

The TSC for the text and image server should be quite similar to the one for the CMFS. The only difference should be that the API for the text server is used. Of course, the scheduling of data which aren't time-sensitive isn't as critical as it is for time-sensitive media, so future developers will also have to decide whether it is worth the effort to implement a TSC for the time-independent media at all.

As was explained earlier, the architecture of the overall system only contains one CMFS. However, the CMFS research team has agreed that multiple CMFSs might be necessary in the News-on-Demand architecture. How this will affect the design of the CMFS is still uncertain. However, it should be relatively easy to modify the TSC to accommodate many CMFSs. The TSC API at the client can send the same scenario to all of the servers, and each TSC can pick out only the objects or streams which concern it (i.e. that are stored on the media server associated with it). This would avoid parsing the scenario, and the other control messages, for each TSC.

6.3.4 The distributed scheduler and user interactions

User interactions such as fast-forward, rewind, resume presentation and quality of service renegotiation during a presentation have not been closely studied yet. It is unclear how the scheduling algorithms will take them into account. Should a new presentation schedule and new stream parameters be derived and sent to the TSC for each of these interactions? Should the client take over the control of the streams from the TSCs? How much scheduling is actually necessary in these cases?

Any change in the delivery parameters for a stream requires the CMFS to be told to stop sending the stream (with `CmfsStop`) and the stream to be prepared with the new parameters (with `CmfsPrepare`) [MHN96]. Also, it must be decided which streams are affected; fast-forward and rewind operations would presumably only involve video, but if there are more than one video clip in the presentation, should they all be scanned forwards of backwards? Should the user choose the individual clip that is to be rewound? Pausing the presentation should obviously affect all of the streams currently being played, and should prevent the presentation of any streams which have yet to be presented - in other words, the presentation and delivery schedules must be "suspended.„ Resuming the presentation requires the positions of the stopped streams to have been remembered so that display resumes at the right point, and streams which haven't yet been presented must also be rescheduled into the future. Renegotiating quality of service parameters during the presentation would presumably involve switching from the streams and objects being presented to another set of objects on different servers, but all the while respecting the positions of the streams when the switch-over occurred, to that the flow of the presentation is disrupted as little as possible. It is clear from these remarks that handling user interactions makes scheduling more difficult. One approach could be to ignore scheduling completely for these tasks, but this would undoubtedly make synchronization of the rest of the presentation more difficult. A preferable approach would be to develop scheduling algorithms which take into account user interactions. For instance, every time that the TSC stops a group of streams, it could be made to remember their positions, so that all that is required is a "resume„ message from

the client. Fast-forwarding or rewinding a presentation could involve deriving a new presentation schedule by the client, which would only include video streams, while the TSC derives a new delivery schedule and the right parameters for playing the streams. Switching servers would obviously require closing all connections on the old servers and sending a new presentation scenario to the new servers with the positions of the streams when they were stopped.

REFERENCES

[BL91] Bulterman, D. C. A. and van Liere, R. (1991). Multimedia Synchronization and Unix. In Proceedings if the 2nd International Workshop on Network and Operating System Support for Digital Audio and Video. (ed. R. G. Herrtwich). pp 109 - 119. Heidelberg, Germany: Springer-Verlag.

[Bur94] 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.

[BG96] Brinskelle, J. and Georganas, N.D. (1996). The University of Ottawa's Synchronization Architecture; Interface Document. Version 2.2. Technical Report, Multimedia Communications Research Laboratory, Department of Electrical and Computer Engineering, University of Ottawa, July 1996.

[BKH+95] Bochmann, G. von, Kerhervé, B., Hafid, A., Dini, P. and Pons, A. (1995). Architectural Design of Adaptive Distributed Multimedia Systems. Technical report, Départément IRO, Université de Montréal.

[BS94] Blakowski, G. and Steinmetz, R. (1994). A Multimedia Synchronization Survey: Specification, Reference Model and Case Studies. In IEEE Journal on Selected Areas in Communication, January 1996.

[FHM+95] Finkelstein, D., Hutchinson, N.C., Makaroff, D.J., Mechler, R. and Neufeld, G.W. (1995). Real Time Threads Interface. User manual. Distributed Systems Group, Department of Computer Science, University of British Columbia.

[Geo96] 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).

[GA91] Govindan, R. and Anderson, D. P. (1991). Scheduling and IPC Mechanisms for Continuous Media. In Proceedings of the 13th ACM Symposium on Operating System Principles, pp. 68 - 80. California, USA: ACM.

[Jar97] Jarmasz, J. P. (1997). Notes on Performance Testing and Comparison of the Centralized and Distributed Schedulers for the CITR News-on-Demand System. MCRLab technical report, Department of Electrical and Computer Engineering, Faculty of Engineering, University of Ottawa.

[LG90] Little, T.D.C. and Ghafoor, A. (1990). Synchronization and Storage Models for Multimedia Objects. In IEEE Journal on Selected Areas in Communications, vol. 8, April 1990, pp. 413-426.

[LGOS96] Li, J.Z., Goralwalla, I.A., Özsu, M.T. and Szafron, D. (1996). Video Modeling and Its Integration in a Temporal Object Model. Technical Report TR 96-02, Laboratory for Database Systems Research, Department of Computing Science, University of Alberta, February 1996.

[Li94] 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 Engineering, Faculty of Engineering, University of Ottawa.

[LKG94.1] 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.

[LKG94.2] 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.

[LLBG96] Lamont, L., Li, L., Brimont, R. and Georganas, N.D. (1996). Synchronization of Multimedia Data for a Multimedia News-on-Demand Application. In IEEE Journal on Selected Areas in Communications, Vol. 14, No.1, Jan. 1996, pp.264-278.

[LLG94] 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.

[LMM94] Leslie, I. M., McAuley, D. and Mullender, S. J. (1994). Operating - System Support for Distributed Multimedia. Technical Report (Pegasus Paper 94-6). Faculty of Computer Science, University of Twente.

[Me96] Mechler, R. (1996). CMFS Data Stream Protocol. Technical Report. Department of Computer Science, University of British Colummbia.

[MES93] Meyer, T., Effelsberg, W. and Steinmetz, R. (1993). A Taxonomy on Multimedia Synchronization. In proceedings of the 4th IEEE International

Workshop on Future Trends on Distributed Computing Systems, Lisbon, Portugal (22-24 Sep. 1993), pp. 97-103.

[MHN96] Makaroff, D., Hutchinson, N. and Neufeld, G. (1996). The UBC Distributed Continuous Media File System. Interface Document, Department of Computer Science, University of British Columbia.

[NMH96] Neufeld, G., Makaroff, D. and Hutchinson, N. (1996). Server-Based Flow Control in a Distributed Continuous Media Server. Paper presented at the 6th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), Zushi Japan, April 1996

[OSEV95] Özsu, M.T., Szafron, D., El-Medani, G. and Vittal, C. (1995). An Object-Oriented Multimedia Database System for a News-on-Demand Application. In ACM Multimedia Systems, 3: 182-203, 1995.

[SE93] Steinmetz, R. and Engler, C. (1993). Human Perception of Media Synchronization. Technical Report 43.9310. IBM European Networking Center, Heidelberg, Germany.

[SN95] Steinmetz, R. and Nahrstedt, K. (1995). Multimedia: Computing, Communications and Applications. Prentice Hall P T R.

[St94] Stallings, W. (1994). Data and Computer Communications, 4th edition, Macmillan, 1994.

APPENDIX A - DATA STRUCTURES

Note: these data structures were created using the C programming language. Most of them are used by the TSC module and the TSC API. They can be found in the file `tsc.h` of the TSC code. The Scenario structure is used by the client application and can be found in the file `scenario.h`.

A.1 Scheduling Structures Used by the TSC

These data structures are used by the TSC to schedule the delivery of data streams by the CMFS to the client. The `Document` structure - a stripped-down version of the `Scenario` structure used by the client - contains sub-structures called `Clips`, which contain information about the scheduling of individual streams, which are identified by their UOIs.

```
typedef struct
{
    /* data from client */
  int     type;/*media type */
  UOI     uoi;
  RttTimeValue duration;

  u_long  serverIp;
  RttThreadId  readerId;
/*identifies the reader to connect to */
    /* filled in by TSC */
  RttTimeValue  deliv_time;  /* delivery time */
  u_long  cid;
  RttTimeValue prepareBound;
} Clip;
```

```
/* Document structure */
typedef struct
{
  /* from client */
  u_int         cliIp;
  int           num_media;
  Clip     stream[ MAX_NUM_MEDIA] ;
  long int skews;
/* not yet impl; from QoS module */

     /* filled in by TSC */
  u_int         in_use;
} Document;
```

The following structures are used by the TSC in its calls to the `CmfsPrepare` interface. `PlayParms` are the parameters which are required by this interface; `PrepParms` contains additional information that the TSC uses to monitor the status of these calls. While `CmfsPrepare` can take a parameter which specifies when a stream is to be delivered, this functionality has not yet been implemented by the CMFS team, and so the parameter, while provided, is not used by `CmfsPrepare` directly.

```
typedef struct
{
  pos      startPos;
  pos      stopPos;
  u_int         speed;
  u_int         skip;
  RttTimeValue startTime;
} PlayParms;

typedef struct
{
  PlayParms     play;
  u_long  cid;
  RttTimeValue delay;
  u_int         compl;
  RttSem  done;
} PrepParms;
```

A.2 New Scheduling Structures Used by the Client

The scheduling data structures used by the client have been modified to allow for stream-level scheduling. The main change is that start times for individual streams have been added. Only the first two layers - the `Scenario` and the `Stream` structures - are given, as the `Segments` structure, already presented in section 3.3.2, has not been changed.

```
/*******************************************************/
/* Stream contains all the information about a media
stream */
/*******************************************************/
typedef struct {
   /* These next vars. will be retrieved from the
database */
   int          media_type;
   p_time duration;       /* in seconds */
   int          num_segments;
   Segments     *first_segment; /* First segment in media
*/
     UOI        uoi;

   /* These next vars. will be filled in dynamically by
Sync. code */
     /*(provided by QOS)*/
   thread_ids   *thread_ids;   /* each stream knows
other thread ids */

     u_long     serverIp; /* IP addr of the server for
the UOI*/
     p_time     start_time;     /* start time of the
stream */
     u_long     cid;
} Stream;


/***********************************************/
/* The Scenario of the multimedia news article */
/***********************************************/
typedef struct {
```

```
    /* These next vars. will be retrieved from the
database */
    int             num_media; /* Number of parallel media's
in article */
    Stream *Stream[MAX_NUM_MEDIA];    /* 1 for each stream
*/

    /* These next set of vars. will be passed by the QoS
module */
    long int     *skews;          /* list of inter-media
skew
                              synchronization values.  Not
                              implemented yet. */

    /* These next vars. will be filled in dynamically by
Sync. code */
    p_time epoch;            /* relative start */
    p_time delta_late;       /* Delta late for entire
presentation */
} Scenario;
```

A.3 Control Messages

The control messages used by the TSC API to communicate with the TSC are presented here. Also given are the status codes that are returned by the TSC to the client and by internal TSC functions.

```
enum TscStatusReturnCode
{
   TSCOK=0, DOCUNAVAILABLE= -1, UOIUNAVAILABLE= -2,
TSCUNABLE=-3, NOSERVER=-4,
PREPFAILED=-5, STOPFAILED=-6, NOTIMPL=-7
};

typedef enum TscStatusReturnCode TscStatus;


     /* MGSs from client to TSC */

typedef struct
{
   Document     doc;
   RttThreadId        cliId;
```

```
} OpenMsg;

typedef struct
{
  u_long dtag;
  PlayParms parms[ MAX_NUM_MEDIA] ;
} PlayMsg;

typedef struct
{
  u_long dtag;
} OtherMsg;     /* misc. msgs. */


     /* MSGs frm TSC to client */

typedef struct
{
  u_long  dtag;
  RttThreadId  TscId;
  TscStatus    status;
} ReplyOpen;

typedef struct
{
  TscStatus status;
} ReplyOther;

     /* The overall message structure*/

typedef struct
{
  u_long type;
  union
   {
    OpenMsg        open;
    PlayMsg        play;
    OtherMsg  other;
    ReplyOpen openDone;
    ReplyOtherdone;
   } parms;
} ControlMsg;


     /*definition of messge types */

#define TSCOPEN      1
#define TSCPLAY      2
#define TSCSTOP      3
#define TSCCLOSE     4
#define TSCREPLOPEN  5
#define TSCREPLOTHR  6
```