Summer 1998

Multiple Streams Synchronization in Collaborative Multimedia Systems

Emilia Stoica
Old Dominion University

Multiple Streams Synchronization in Collaborative Multimedia Systems

by

Emilia Stoica
M.Sc. Polytechnical University of Bucharest, Romania, 1989

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements of the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
July 1998

Approved by:

Dr. Hussein Abdel-Wahab (Director)

Dr. Kurt Maly (Member)

Dr. Ravi Mukkamala (Member)

Dr. Stewart Shen (Member)

Dr. Jean-Philippe Favreau (Member)

Abstract

Multiple Streams Synchronization in Collaborative Multimedia Systems.

Emilia Stoica
Old Dominion University, 1998
Director: Dr. Hussein Abdel-Wahab

With the recent increase of the communication bandwidth and processor power, new types of applications have emerged. Among them, there are multimedia applications, in which users are able to control, combine, and manipulate different types of media, such as text, sound, video, computer graphics, and animation. A key requirement in any multimedia application is to synchronize the delivery of various media streams to the user. To achieve this, the sender has to provide the temporal relations between the streams as they are captured. Since the receiver uses this information in streams presentation, its accuracy is very important.

Our main contribution is to provide a suit of synchronization algorithms for audio, video and X-windows streams that work correctly in the presence of load variations. First, we propose a mechanism for assigning a correct synchronization specification to media units that takes into account the workload variation at the sender: although this issue is critical, it has been largely ignored in previous work. Second, for detecting the skew between the streams, we propose a synchronization condition that works in the general case of streams having different media unit durations. Based on this condition, we develop an adaptive lip-synchronization algorithm. By estimating the display time of video frames, our algorithm is robust and stable in the presence of both network and workstation load. To synchronize the X-windows stream with the audio/video stream we propose a novel approach that combines dropping X packets with delaying the X client. Finally, we extend our algorithms to a

distributed environment. We do this by proposing (1) a mechanism for extracting the synchronization information from mixed audio streams, and (2) a lightweight mechanism to achieve global clock synchronization.

To my parents.

Acknowledgments

I would like to thank very much to my advisor, Professor Hussein Abdel-Wahab for his valuable suggestions, continued guidance and support in the preparation of this thesis. I am also very much indebt to Professor Kurt Maly, for his permanent encouragement and for many useful comments he gave me while this work was carried out.

I want to thank from my heart to my parents who taught me how important it is to learn and to always be as best as I can.

I am very confident that without the loving support of my husband, Ion and the patience of our son, George, I could have not done this work.

"The voyage of discovery is not in seeking new landscapes but in having new eyes."

Marcel Proust

Table of Contents

List of Tables xi

List of Figues xiii

I Introduction 1

I.1 Issues . . . 3

I.1.1 Media Synchronization Specification . . . 3

I.1.2 Media Display Time . . . 5

I.1.3 Synchronization Condition . . . 6

I.1.4 Lip-Synchronization . . . 7

I.1.5 Synchronization of the Shared Windows Stream . . . 8

I.1.6 Media Synchronization in Distributed Systems . . . 9

I.2 Objectives . . . 11

I.3 Experimental Setup . . . 12

I.4 Outline . . . 12

II Related Work and Motivation 14

II.1 Media Synchronization Specification . . . 15

II.2 Media Display Time . . . 15

II.3 Synchronization Condition . . . 16

II.4 Lip-Synchronization . . . 17

II.5 Synchronization of the Shared Windows . . . 23

II.6 Synchronization in Distributed Systems . . . 25

II.7 Motivation of Work 27

III Effect of Workstation Load **30**

III.1 Exploring Real-time Capabilities 31

III.1.1 Experimental Design 31

III.1.2 Measurements 34

III.1.3 Results Interpretation 34

III.2 Media Synchronization Specification 36

III.2.1 Acquisition of Continuous Streams 37

III.2.2 The Mechanism of Sharing X-Windows 38

III.2.3 Specification for Continuous Streams 39

III.2.4 Specification for the Shared Windows Stream 42

III.3 Media Display Time 44

III.3.1 Estimation for Continuous Streams 46

III.3.2 Estimation for the Shared Windows Stream 50

III.4 Summary 53

IV Synchronization Algorithms **55**

IV.1 Synchronization Condition Between Streams 55

IV.2 The Lip-Synchronization 58

IV.2.1 Implementation Issues 62

IV.3 Synchronization of the Shared Windows Stream 63

IV.3.1 Key Considerations 63

IV.3.2 The Synchronization Algorithm 65

IV.4 Summary 70

V Media Synchronization in Distributed Systems **72**

V.1 Extracting the Synchronization Information from Mixed Audio Streams 73

V.2 A Common Time System for a Multimedia Application 77
V.3 Summary 82

VI Effect of Network Load **84**
VI.1 Lip-Synchronization 85
VI.1.1 Experiment Description 85
VI.1.2 Results and Evaluation 88
VI.2 Synchronization of Shared Windows 92
VI.2.1 Experiment Description 92
VI.2.2 Results and Evaluation 93
VI.3 Summary 95

VII Results and Conclusions **97**
VII.1 Media Synchronization Specification 97
VII.2 Media Display Time 98
VII.3 Synchronization Condition 99
VII.4 Lip-Synchronization 100
VII.5 Synchronization of the Shared Windows Stream 100
VII.6 Extension to a Distributed System 101
VII.7 Future Work 101
VII.8 Impact of Contribution 102

References **103**

Appendix A Classification of X Requests **110**

Vita **111**

List of Tables

Table		Page
III.1	Variation of the inter-arrival time [ms].	32
III.2	Effect of real-time scheduling.	32
III.3	Notations.	40
III.4	The RTT time for a Unix socket in the presence of various loads.	45
IV.1	Specification of lip-synchronization protocols.	62
IV.2	Specification of X-windows synchronization protocols.	67
VI.1	Percentage of audio and video frames successfully delivered at the destination in the presence of heavy network load.	86
VI.2	Percentage of video frames skipped with protocols P2 and P3.	86
VI.3	Evaluation of the asynchrony between audio and video in the presence of heavy network loads [number of audio frames].	87
VI.4	Evaluation of the asynchrony between audio and X windows in the presence of heavy network loads [number of audio frames].	93
A.1	X Requests that crash the X client if dropped.	111
A.2	X Requests that crash the X client if dropped (cont.)	112
A.3	X Requests that freeze the X client if dropped (queries).	113
A.4	X Requests that freeze the X client if dropped (cont.)	114
A.5	X Requests that affect other X clients if dropped.	114
A.6	X Requests that can be safely dropped.	115
A.7	X Requests that can be safely dropped (cont.)	116

A.8 X Requests that can be safely dropped (cont.) 117

List of Figures

Figure Page

I.1 A collaborative multimedia application integrating audio, video and shared windows. 3

I.2 Effect of mixing audio frames on the temporal synchronization problem. 9

III.1 The video inter-arrival time variation when video, audio and the following job was running: (a) none, (b) read from disk, (c) print on the console, (d) twenty busy processes, (e) random memory write and (f) Mosaic, move windows on the screen. 33

III.2 The video inter-arrival time variation in real time when the following job was running : (a) twenty busy processes (b) Mosaic, move windows on the screen. 34

III.3 The mechanism of sharing X clients using *XTV*. 38

III.4 Effect of load on the display time of a video frame when: (a) no other load was introduced in the system, (b) the window was sometimes moved, (c) a busy process was concurrently running, and (d) another video image was displayed. 47

IV.1 Intuitive interpretation of the model (a) ideal case, (b) when video is ahead, (c) when video is late. 56

V.1 The packet queue and the values of *lastDequedAudioPacket* and *lastStreamPacket* variables for two audio streams at three time instances. 74

V.2 The time diagram for evaluating the starting time. 78

VI.1 The Network configuration. 85

VI.2 Variation of the skew between audio and video with protocol P1 (no correction), when a load of (a) 8 Mbps, (b) 8.25 Mbps, (c) 8.5 Mbps and (d) 8.75 Mbps was put on the network. 89

VI.3 Variation of the skew between audio and video with protocol P2 (skip a late video frame, delay an early video frame), when a load of (a) 8 Mbps, (b) 8.25 Mbps, (c) 8.5 Mbps and (d) 8.75 Mbps was put on the network. 90

VI.4 Variation of the skew between audio and video with protocol P3 (delay an early video frame, delay audio if it is a trend for video to be behind, no video skip), when a load of (a) 8 Mbps, (b) 8.25, Mbps, (c) 8.5 Mbps and (d) 8.75 Mbps was put on the network. 90

VI.5 Variation of the skew between audio and video with protocol P4 (no video skip, delay video if it is behind, delay audio if it is a trend for video to be behind), when a load of (a) 8 Mbps, (b) 8.25 Mbps, (c) 8.5 Mbps and (d) 8.75 Mbps was put on the network. 91

VI.6 Variation of the skew between audio and the X windows stream with protocol X1 ((a), (b), (c)) and with protocol X2 ((d), (e), (f)) when a load of (a) and (d) 6 Mbps, (b)and (e) 7 Mbps, (c) and (f) 8 Mbps was put on the network. 94

VI.7 Variation of the skew between audio and the X windows stream with protocol X3 ((a), (b), (c)) and with protocol X4 ((d), (e), (f)) when a load of (a) and (d) 6 Mbps, (b)and (e) 7 Mbps, (c) and (f) 8 Mbps was put on the network. 94

Chapter I

Introduction

"Tell me and I'll forget; show me and I may
remember; involve me and I'll understand".
Chinese proverb

Recent advances in computer and network technologies have made feasible a new generation of distributed applications, such as videoconferences, distance learning, and tele-medicine*. These applications integrate different information media: audio, video and data; therefore they are called multimedia applications.

Collaborative multimedia applications provide users with more than audio, video and data; they also provide a shared workspace, which is comprised of text, graphics and drawings [31, 32, 34]. Providing audio and video enables participants to communicate verbally and visually on a task. Providing the shared workspace enables participants to have the same view of the shared windows on their screen.

Figure I.1 shows the interface of IRI [32], a collaborative multimedia application developed at Old Dominion University. IRI is used for teaching classes when students are situated geographically apart from each other. In this instance, the teacher and two students are involved in a discussion regarding an AUTOCAD tool.

*The thesis used as journal model the article "Using Timed CSP for Specification Verification and Simulation of Multimedia Synchronization", *IEEE Journal of Selected Areas in Communications*

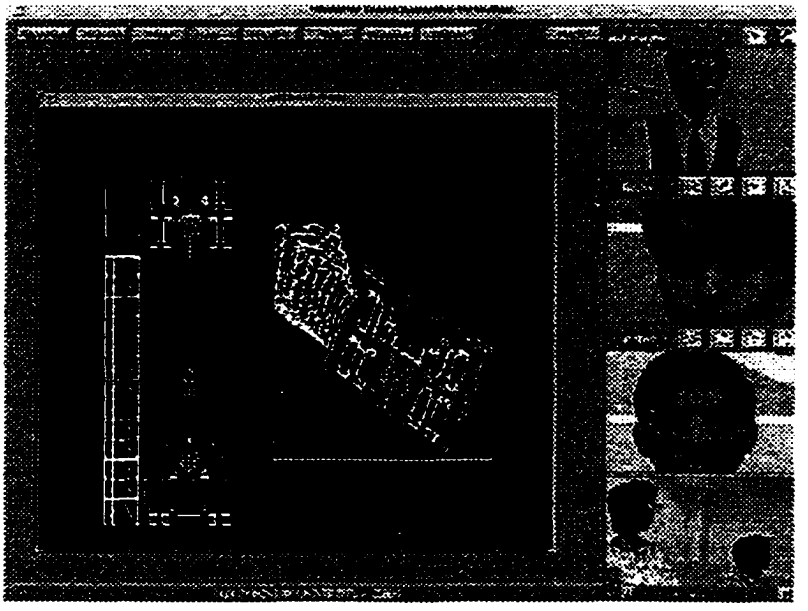


Figure I.1: An instance of IRI multimedia application interface.

The system captures the audio and video streams originating from teacher's and students' machines and presents them on each workstation. Because the teacher has started auto-cad, the corresponding window appears on every student workstation. In addition, the teacher's interaction with auto-cad is visible to each student through the mechanism of sharing windows.

A critical issue that any multimedia system has to address is how media streams are synchronized when they are played to the end users. In this context, multimedia synchronization can be defined as the task responsible for the temporal coordination and presentation of multimedia objects.

At the source, there is a specific temporal relation between the streams. At the destination, this temporal relation needs to be preserved during the presentation. As an example consider the IRI application. The teacher's workstation (source) establishes the temporal relation between his audio, video and auto-cad interaction. This temporal relation needs to be preserved by the audio, video and the shared

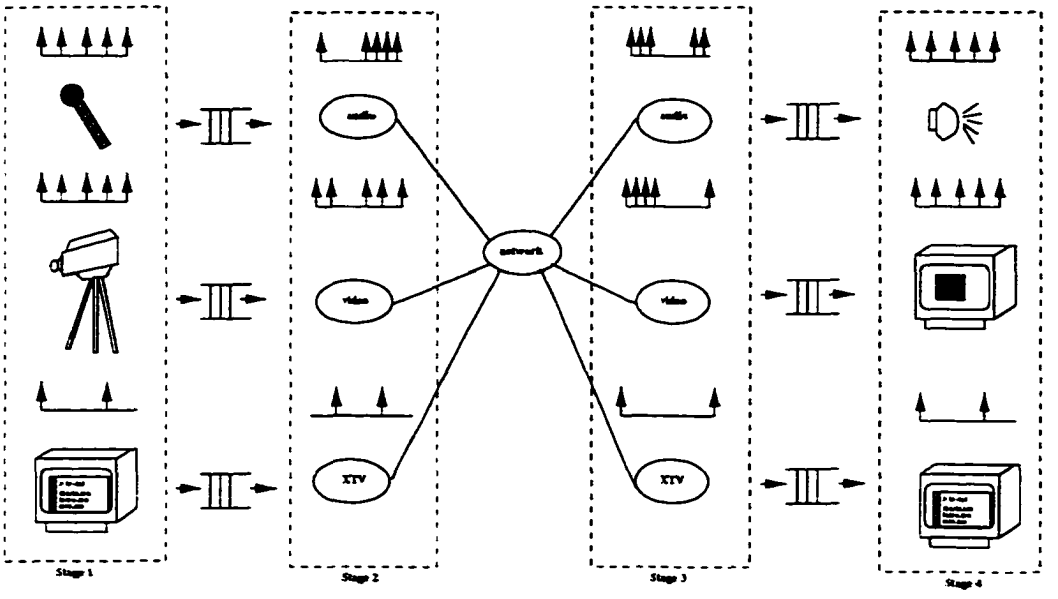


Figure 1.2: A collaborative multimedia application integrating audio, video and shared windows.

windows processes on each student workstation (destination).

Our work focuses on providing fine-grain synchronization of audio, video and shared windows streams in a collaborative multimedia system. To ensure portability, we design our synchronization algorithms to work on top of the existing transport protocols such as UDP or RTP [39].

Although previous related work [17, 23] used real-time networks and operating systems as a solution to achieve high-quality presentations, in our work we concentrate on best-effort systems. We made this decision for two reasons: first, algorithms designed for non real-time systems can also work in real-time ones; second, from our experience, there are many situations when the real-time extensions of the *current* operating systems (e.g., Solaris 2.5) do not offer significantly better performance than the traditional time-sharing policies [2].

I.1 Issues

Figure I.2 shows the software architecture of a typical multimedia application. Audio frames are captured by the microphone, queued by the audio device driver, read by the audio process, sent over the network to the destination application, queued again by the audio process, and played by the speaker. Similarly, video frames are captured by the camera, queued by the video device driver and read by the video process. After that they follow the same path to the destination. Shared windows are generated by an X client, captured by the data sharing process, sent over the network to the destination, and then sent to the local X server.

The temporal synchronization problem poses the following issues: (1) assign the synchronization information, (2) estimate the display time of media units, (3) assign a synchronization condition, (4) design a lip-synchronization algorithm, (5) integrate the shared windows stream and (5) extend the solution to a distributed system. In the following, we present in detail each of these issues.

I.1.1 Media Synchronization Specification

There is a temporal relation between audio, video and the shared windows media units[†] when they are captured. This temporal relation is called *synchronization specification*. The synchronization specification is used by the destination application to present the streams. For example, video frame *3*, audio frame *3* and the shared windows packet that displays an image are all generated simultaneously by the microphone, video camera and the X client. If this synchronization information would be incorrect, it would be impossible to accurately synchronize the streams at the receiver.

Ideally, the temporal relations between the media units at generation time

[†] A media unit can be an audio frame, a video frame, or a shared windows packet.

(Figure I.2, stage 1) are preserved exactly when the media units are transferred to the source application (Figure I.2, stage 2). In reality, due to the nondeterministic nature of the today's mainstream operating systems, the synchronization specification perceived at the application level, may be different from the real one, which is determined when the streams are captured. This is due to the fact that in a general-purpose operating system, it is fairly difficult to schedule processes at regular time intervals, as they compete with other processes for CPU.

Existing solutions ignore this issue: they generally determine the synchronization specification based exclusively on the time when media units arrive to the application [4, 6, 7, 12, 16, 18, 20, 24, 28, 35, 34, 43, 45, 3]. For example, in RTP [39], each audio and video packet has a temporal timestamp which indicates the time the packet has been received by the source audio or video process.

We show how load variation at the source can lead to an incorrect synchronization specification, and describe a robust solution to this problem. Our mechanism for a synchronization specification is flexible enough to be incorporated in almost any temporal synchronization solution, while also substantially improving the quality of the presentation at the destination.

In addition, we show that the immediate solution for scheduling multimedia processes in real-time is not always successful because even if the operating system is fully preemptive, the X windows process is not [2].

I.1.2 Media Display Time

To ensure a high quality presentation, the destination application has to schedule the media units according to the synchronization information. However, merely simultaneously transmitting two frames (e.g., audio and video) to their presentation devices, does not guarantee that they will be played at the same time. This is due to various factors, such as kernel buffering and processor scheduling policy, that may

introduce a non negligible delay between the time when a media unit is scheduled by the application and the actual time when it is played by the presentation device (e.g., speaker). We call this interval *display time*.

In the audio case, the display time consists of the queuing delay associated to the device driver buffer. For video and shared windows, the display time has to take into account the fact that the video images/shared windows packets are displayed by another process, i.e., the X server. The display time consists of both the queuing delay associated with the X server buffer [46], and the time interval created while the X server process waits to be scheduled.

Two media units which are simultaneously sent to their presentation devices play at the same time if and only if their display times are equal. Since in practice this is not the case, it is necessary to take into account the media display times in order to correctly synchronize the media units. The effect of the media units' display time on temporal synchronization has been partially considered by Elefteriadis [16], and Owezarski [42]. While Elefteriadis accounts for only the display time of audio frames, and neglects the display time of video frames, Owezarski assumes that the display time is the same for both audio and video frames, which greatly simplifies the problem.

In Chapter III we show the importance of differentiating between the audio, video and shared windows display times and propose a set of algorithms that take into account these times.

I.1.3 Synchronization Condition

Usually, a synchronization algorithm defines a condition that streams should meet in order to be synchronized. This is called *synchronization condition*. Examples of synchronization conditions are: (1) media units with the same sequence number should play simultaneously [11], (2) the difference between the acquisition timestamps

of the master and the slave[‡] frames should be smaller than the accepted asynchrony between the streams [12, 13, 16, 24, 43, 45, 49, 3], and (3) streams should all reach a synchronization point in order to play [33].

Note that in these examples, the second condition requires timestamps, which represent redundant information since frames are already assigned sequence numbers in order to detect network losses. The first condition assumes that the streams to be synchronized have media units with the same duration[§]. Similarly, the third condition assumes that the frame durations have a common divisor. These restrictions make the solutions based on these conditions quite inflexible. For example, using these synchronization conditions makes it very difficult, if not impossible, to arbitrarily change the audio frame sizes at run-time in order to optimize the transport protocol (see [23] for such optimization).

We address these problems in Chapter IV, where we propose a new synchronization condition that can handle streams with arbitrary media unit duration, and yet not waste the network bandwidth.

I.1.4 Lip-Synchronization

The purpose of a lip-synchronization[¶] mechanism is to overcome the delays introduced by the network and the operating system. This is usually achieved by relying on interprocess communication mechanisms to coordinate media unit presentation based on the relative progress of the streams. The two streams are synchronized by dropping video frames if video is late or pausing the video stream if it is ahead [4, 6, 7, 8, 9, 11, 12, 13, 16, 18, 20, 24, 28, 43, 49, 60].

[‡]A master stream is usually played without any of its frames to be delayed or dropped; on the other hand, the frames of the slave stream are delayed or dropped if needed in order to match master stream frames.

[§]For periodic streams, the media unit duration is equal to the stream period.

[¶]The synchronization of audio and video is called lip-synchronization.

From our experience, the "drop-delay video" approach works fine for 320×240 pixels, 24 bits depth windows, but it does not always work for 640×480 windows, when the display of a video frame takes up to 250 ms. When the "drop-delay video" approach is used, the image freezes frequently as a result of many video frames being dropped.

Our lip-synchronization algorithm does not drop any video frame. The synchronization is achieved by estimating the display time of video frames and delaying audio when silence periods are detected.

Obviously more hardware resources such as memory, better video boards and faster machines may significantly improve the behavior of the algorithms. For example, from our experience in the IRI project, in fall of 1997, while running IRI without any synchronization mechanism, there was an average of 250 ms skew between audio and video and the presentation was visibly annoying. After the machines were upgraded from 75 MHz to 100 MHz, under the same conditions, there was no observable skew between the streams. Does this mean that we need to ignore the lip-synchronization issues and consider them to be problems which can be solved by new or better hardware ? In our opinion simply increasing hardware resources is not an acceptable solution. There are still cases of transient overload, such as when a large postscript file is displayed, that needs to be handled correctly. In addition, a complete algorithm would permit the use of old workstations with good results. Thus, our approach is to identify the key issues for lip-synchronization and to develop mechanisms that efficiently utilize any existing resources.

I.1.5 Synchronization of the Shared Windows Stream

Audio is a periodic, stateless stream. Video is a periodic, stateful stream, because it explores temporal redundancy and models a picture as a translation of the picture at a previous time (e.g. in CellB the current picture is expressed as pixels difference from

the previous one). However, even if a video frame is dropped, the video application does not crash. On the other hand, the shared windows stream is a stateful and aperiodic stream. A request usually depends on the previous requests. For example, a request to create a window is related to the previous request which creates the parent window. If audio and video media units can be dropped in order to keep the streams synchronized, a shared windows request can be dropped only if we are sure that no subsequent request will refer to it; otherwise the application may crash.

The difficulties in synchronizing the shared windows stream are due to both (1) its stateful character, and (2) the large display times of some requests[‖], which require putting an image or filling a rectangle. In addition, the type of a request does not necessarily say how long its display time is. For example, the display time for the request that displays an image (*PutImage* [46]) on the screen is around 13 ms for the maximize/minimize/close window bitmap, and up to 475 ms for a three square inches color picture.

So far, the existing solutions either delay audio when the shared windows stream tends to be behind [35], or change the rate of the shared windows stream to catch up with the other streams [33]. From our experience, in a real-time video conference where the shared X clients load pages with heavy graphics, the shared X windows stream is far behind the audio stream (6-7 seconds). This is due to the cumulative effect of large display times of the shared windows packets. In this situation, delaying audio makes the presentation very annoying. Adapting the sending rate of the shared windows stream is somewhat ineffective given that the rate of playing the shared windows requests depends on the X server processing rate. In many cases, such as performing heavy window updates, this rate lags significantly behind the audio. As a result, these solutions are not adequate under heavy shared windows traffic.

[‖] We assume that each media unit corresponds to exactly one shared windows request.

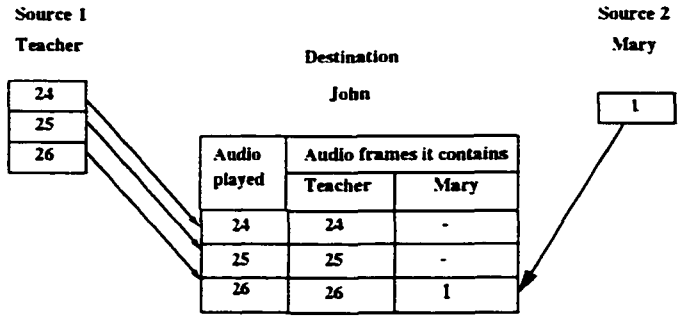


Figure I.3: Effect of mixing audio frames on the temporal synchronization problem.

Our solution to synchronize the shared windows stream with continuous streams, such as audio and video, is to identify the requests that can be dropped and to drop them when the shared windows stream is behind. In addition, if this is not enough, we can delay the X client that generates the requests until the receiver's X server catches up. In practice, this algorithm proved to be robust in the presence of very heavy shared windows traffic.

I.1.6 Media Synchronization in Distributed Systems

In the case of a multiparty application, an additional problem is caused by the fact that when more than one participant speaks at one time, incoming audio streams need to be mixed at the destination before they are played. As a result, the synchronization information is lost.

To better understand this problem, consider the example of a session with one teacher and two students, John and Mary (see Figure I.3). Initially, assume that only the teacher speaks. Consequently, the audio process on John's workstation will receive the teacher's audio frames and send them to the audio device. The audio device maintains a counter of the frames played so far. As long as the teacher is the only one who is talking, there will be a one-to-one correspondence between

the sequence number assigned by the teacher to his audio frames and the sequence numbers assigned by John's audio device to the frames it plays.

Assume that after 25 audio frames from the teacher are played, Mary starts to speak too. Then the 26^{th} audio frame played by John's audio device will now contain the 26^{th} teacher's audio frame and Mary's first audio frame. Video and audio streams originating from each sender (the teacher and Mary in our case) should be synchronized among themselves. In the teacher's case this is quite easy, since a request to the audio device will give the correct sequence number 26 of the frame which is currently playing. However, this is not true for Mary. When her first video frame plays, a request to the speaker returns audio frame 26 as the current playing audio frame. If this is interpreted as her current audio frame, that is audio frame 26, then all of the video frames coming from Mary will be dropped.

Our literature search indicates that the issue of maintaining the correct synchronization information of mixed audio streams has been ignored in previous work. We address this issue in Chapter V, where we propose a simple mechanism which maintains the list of the audio frames sequence numbers that are mixed in each audio frame sent to the audio device. This way the synchronization information from multiple sources is preserved.

A side issue that needs to be addressed in the context of a distributed system is the common time at all workstations. This is useful if the application is recorded and played back, since it provides a global order of events in the system. Our motivation to investigate this issue was the requirement that IRI application needs to be recorded and played back. There are numerous solutions in literature for this problem, among of which are the following. One solution is to use the service provided by the U.S. National Institute of Standards and Technology (NIST) [61]. Unfortunately, although this service is accessible through a regular modem, it is not suitable for a large population of clients trying to access it simultaneously. Another solution is to use

the Unix time daemon *timed*, which is based on an elected master host to measure offsets of slave hosts and to send periodic corrections to them [19]. Similarly, the solutions proposed in [36], [45] and [3] assume a master workstation that provides the time. A drawback of these solutions is that the master workstation represents a single point of failure. In addition it can be a bottleneck in the presence of a large number of workstations. As an alternative, we propose a lightweight scheme that assumes no dedicated time servers and no dedicated hardware. We note that at the time we developed this solution [56], we have learned that a similar one is used by the OSF Distributed Computing Environment [47].

I.2 Objectives

In this thesis we study and develop a set of mechanisms that ensure synchronization support for distributed multimedia applications which integrate audio, video and the shared X-windows stream.

Our objectives are the following:

1. provide a correct synchronization specification at the sender
2. account for the display time at the receiver
3. design a synchronization condition
4. design the synchronization algorithms
5. extend our algorithms to a distributed system. To achieve this we need to
    - extract the synchronization information from mixed audio streams
    - provide a common time for a distributed multimedia application

I.3 Experimental Setup

To verify and validate our algorithms, we used the Interactive Remote Instruction (*IRI*) project [32], developed at the Computer Science department of Old Dominion University. *IRI* requires synchronization support in a distance learning multimedia application where parties use best-effort operating systems and networks.

The experiments in this thesis used SPARC 5 workstations, with 32 Mb of memory, running Solaris 2.5 and equipped with Sun audio and video devices. The workstations are interconnected by a Switched Ethernet (3ComLinkSwitch1000) which basically creates a dedicated 10 Mbps link between any two workstations. We captured the audio and video of the teacher sitting at a workstation and played the streams on another workstation. Video frames (640 × 480 pixels) were CellB [59] hardware compressed, software decompressed and displayed in an 8-bit depth window. The media unit duration of an audio frame was 64 ms, while the media unit duration of a video frame was 100 ms.

I.4 Outline

The thesis is organized as follows. Chapter II describes work related to each of the issues under consideration. In Chapter III we show why real-time is not a suitable solution for the temporal synchronization problem. We also introduce our synchronization specification and mechanisms for estimating the display time of audio, video and shared windows streams. In Chapter IV we describe our lip-synchronization algorithms, while in Chapter V we introduce a complete solution for synchronizing audio, video and the shared windows stream. Chapter VI presents experimental results and the evaluation of our protocols. Finally, Chapter VII summarizes the contributions and applications of our work.

Chapter II

Related Work and Motivation

"The important thing is not to stop questioning".

Albert Einstein

During the past few years, a large number of services, protocols and mechanisms have been developed to meet the synchronization requirements in both local and distributed networks. Our work relates to research in (1) media synchronization specification, (2) media display time, (3) synchronization condition, (4) lip-synchronization, (5) synchronization of the shared windows stream, and (6) extension to a distributed system.

In this chapter we describe the most representative work in the temporal synchronization field and the motivation of our work. We begin by presenting a solution for assigning a correct synchronization specification to media units and two solutions for estimating the display time. Next, we describe the synchronization conditions widely used in literature and the lip-synchronization solutions that use them. We continue by presenting two algorithms that synchronize the shared windows stream with audio and video. Finally, we describe two synchronization algorithms that achieve a global clock in a multimedia system.

II.1 Media Synchronization Specification

A multimedia process timestamps each frame. Ideally, the timestamps assigned by the source application reflect the same temporal relation between the streams as the temporal relation when the streams were captured. In the presence of workstation load, the times when media units arrive at the application greatly vary and consequently the synchronization specification assigned by the application may be wrong. As this is used by the destination application to synchronize the streams, the whole presentation may be annoying.

A solution to this problem is to discard every frame that arrives after its deadline [13]. For example, for a 30 frames/sec video rate, the deadline is 33 ms after the deadline of the previous frame. In this situation, even if two temporally related audio and video frames arrive late at the source application, they are both discarded. As frames may also be discarded by routers while being sent over the network, the approach may result in too many and unnecessary discarded frames. Our policy is that only the destination application discards frames in order to achieve synchronization. Therefore, in our work, we assign a synchronization specification by estimating the correct time a media unit has been generated.

II.2 Media Display Time

After media units arrive at the destination, the application presents them to the user. The variable delays caused by the operating system and the presentation devices may lead to situations that two media units sent at the same time to their presentation device, actually end up playing at different times. Depending on the difference between the times the media units are actually visible to the user, the presentation may be in sync or not.

Elefteriadis [16] proposes a mechanism that estimates the display time of an

audio frame based on audio device buffer occupancy. To find the sequence number of the currently playing audio frame, the system keeps a finite history of received audio frames. The audio frame k that currently plays, satisfies the condition

$$\sum_{i=k}^{l} L(a_i) \geq O(t_{v_j}) > \sum_{i=k+1}^{l} L(a_i) \qquad \text{(II.1)}$$

where $L(a_i)$ denotes the length of the i-th audio frame in samples, l is the most recent audio frame received and $O(t_{v_j})$ is the kernel audio buffer occupancy (in samples) when video frame j was received. The display time of a video frame is ignored. Owezarski [42] assumes that the display time is the same for both audio and video frames, which greatly simplifies the solution.

In our work we show how important it is to account for the display times of video and the shared windows streams and we provide appropriate solutions.

II.3 Synchronization Condition

The synchronization condition is the condition for presenting the media units to the user. If the synchronization condition is satisfied, a media unit is played, if not, resynchronization is required.

Widely used in literature are the timestamps [4, 6, 7, 12, 16, 18, 20, 24, 28, 35, 34, 43, 45, 3], sequence numbers (if media units have the same duration) [11, 13, 49], and synchronization events [8, 9, 60]. For the timestamps and the sequence numbers, the synchronization condition requires that two media units with the same timestamp or sequence number to be presented at the same time. In the case of synchronization events, the synchronization condition states that two media units are presented when they both reach the same synchronization event. There are also approaches that use Petri nets [21, 30], dedicated languages, like Smil [22] and Esterel [14], and grammars [44], where special constructs state the conditions the streams need to

satisfy in order to be synchronized.

Both the approaches based on sequence numbers and synchronization events restrict the streams to be in a special relation, precisely, their media unit durations to have a common divisor. For example, for a 30 frames/sec video stream, the media unit duration of the audio stream should be a multiple of 33 ms, in order to assign synchronization events, or 33 ms in order to assign sequence numbers, with the existing solutions. This restricts very much the application, as usually audio has a higher rate than video, in order to minimize delays. The solutions that use timestamps, waste network bandwidth, as packets are already assigned sequence numbers in order to detect network losses.

In our work we suggest a simple mechanism that allows streams with different media durations to be synchronized, uses sequence numbers in order not to waste network bandwidth and does not require any special language or grammar construct, thus making it easy to be integrated with any other application.

II.4 Lip-Synchronization

Audio and video streams impose tight temporal constraints. A presentation is considered to be in the user desirable range as long as the skew between the two streams is within (-80, +80) ms [54]. However, a skew between (-160, +160) ms, although visible, is not annoying. There have been many synchronization proposals in the last few years. The most representative are as follows:

ACME Server [4] developed at the Massachussets Institute of Technology assumes a real-time operating system. The algorithm uses a logical time system (LTS) that can be device, connection, or clock driven. For example, in a multimedia conferencing system, the LTS is connection driven; each stream maintains its LTS and its current time. For a multimedia document browser, the LTS is device driven; each stream

keeps track of its current time, but there is only one LTS for all the streams, driven by the device of the master stream (e.g., the audio device). The LTS is incremented every time period of the stream (if device or connection driven) or of the clock (if clock driven). For example, for a 30 frames/sec video rate, it is incremented every 33 ms. The current time is incremented when a frame has arrived. To keep the LTS and the current time in sync, frames may be dropped or duplicated.

Athena Muse [20] developed at the Massachussets Institute of Technology uses a time dimension where streams are attached to. No two components are tied to each other, making easy to add, remove channels. A time dimension has a current position in its range, updated by signals. User-interface controls (scroll-bars and command buttons) or the system clock can generate the signals. Interstream synchronization is achieved by keeping each stream in sync with the time dimension (making an analogy with the ACME Sever [4], we can view the time dimension as an LTS which is device driven.)

Xphone [16] is a multimedia communication system developed at Columbia University. It provides synchronized playback of audio and video locally or across a network. At the sender, audio and video frames are timestamped. At the destination, an audio frame is immediately played, while a video frame is played if the following condition is satisfied:

$$t^a_{a_{k-1}} \leq t^a_{v_j} < t^a_{a_k} \qquad \text{(II.2)}$$

where $t^a_{a_k}$ is the acquisition time of audio frame k (that is currently playing) and $t^a_{v_j}$ is the acquisition time of video frame j (the last one received). If $t^a_{v_j} < t^a_{a_{k-1}}$, then the video frame is dropped. If $t^a_{a_k} \leq t^a_{v_j}$, then the video frame is queued.

Continuous Media Player [49], developed at Berkeley University is a system that

plays audio and video on UNIX workstations. Audio frames have higher priority and are played as soon as they arrive at the destination. Video frames have associated an *earliest start time* and a *latest start time*. Frames that arrive within these two times are played. A late video frame is dropped, an early frame is delayed. The player uses an adaptive feedback algorithm to match packet flow to the available resources. Every 300 ms, it computes a penalty of 10 points if a video frame is dropped or lost in the network. If two consecutive frames are dropped, the penalty is still 10 points. The player uses the penalty to adjust the current frame rate as follows: $current_{Rate} = current_{Rate}(1 - penalty/100) + min_{Rate} \times penalty/100$. If the penalty is 0, no adjustment is made. If the penalty is between 0 and 100, the current rate is reduced. If the penalty is 100, the current rate is set to a minimum rate.

Recently, **Qiao and Nahrsted** [43] from the University of Illinois at Urbana-Champaign, have designed a fine-grain lip-synchronization algorithm for best-efforts environments. At the end of the decoding time of an audio frame, the decoding time of the corresponding MPEG video frame is estimated, by averaging over previous values. The video frame is decoded only if its decoding time is smaller than the difference between the play time of the video frame and the play time of the audio frame (+80ms). An I type frame is decoded and played even if late, unless only I type frames are left in the down stream. A P type frame is decoded and played unless it is the last one before the next I frame. After late I or P frames are played, subsequent B frames are skipped to catch up.

The **MultiSync model** [12] developed at National Taiwan University assigns higher priority to most important media (e.g., audio) and lower priority to other media (e.g. video, text). The highest priority stream is played continuously, while the lower priority streams adopt a delay-or-drop policy. Interstream synchronization is ensured by an absolute synchronization of each media with a time axis. The video process

uses three timestamps – *start time*, *end time* and *current time* – to check whether a frame should be played or not (*start time* and *end time* represent the beginning and the end play times for the video frame, while *current time* is the time at which the frame has been received by the video process). If the current time is between the *start* and *end* times, the video frame is played. If it is greater than the *end time*, the frame is dropped and if it smaller than the *start time*, the frame is delayed.

Fujikawa *et al.* [18] from the University of Taiwan, suggest a mechanism based on streams rate monitoring. The presentation consists of a group of objects, where each object may comprise audio, video and text. The play time of each media unit of an object is an offset from the time the object started. For example, assume that an object consisting of audio and video starts at 5:00. The offset for the first audio frame is 0 and the offset for the first video frame is 2 minutes. Audio will start playing at 5:00 and video will start playing at 5:02. The presentation may be delayed or accelerated by modifying the start time of the streams, and thus the absolute playing time of its units. Using the previous example, if the video stream is 2 seconds late, then, its start time is modified to be 4:58. If video stream is 2 seconds early, its start time is modified to 5:02.

Blair et al. [8] have designed an object-oriented platform that can be used for both intra and interstream synchronization, using the parallel programming language Esterel and a modified version of the Chorus real-time microkernel. An Esterel program consists of a set of parallel processes that execute synchronously and communicate with each other by signals. As an application of the platform, they present how synchronization for audio and video can be achieved. There are three objects : audio (A), video (V) and a coordinator (R). Whenever an audio/video frame arrives from the audio/video device, A/V sends a signal to R and waits for a signal from R that tells when to play the audio/video. A also sends to R a signal a_{reqd} which encapsu-

lates a hardware interrupt when the requested audio data presentation is over. When R receives an a_{reqd} signal from A, it computes the next ideal time for an audio frame and signals A to play an audio frame as soon as it comes. Thus, audio will play continuously, while video checks for interstream synchronization. When R receives a signal from V indicating that a frame has arrived, it computes the ideal time for that frame and if the time difference between the ideal time for the last audio frame and the ideal time for the video frame is greater than 100 ms, it emits a signal. The application may react to this signal by lowering the transmission rate.

Correia and Pinto [13] from the University of Portugal, have done the only work we are aware of that takes into consideration the effect of workload variation at the transmitter on streams synchronization. Their solution is to drop a frame that has arrived late at the application. The next frame will carry an indication of this action. The interstream synchronization mechanisms assumes that the streams have the same media unit duration. Each media unit has associated a reception timestamp. If the difference between the reception time of master unit n and the reception time of slave unit n is greater than a threshold, then the master stream is delayed. This mechanism is implemented for each master-slave pair.

Biersak et al. [7] from Institut Eurecom, France, have developed a scheme for the continuous and synchronous delivery of stored multimedia streams, when a stream is distributed over multiple server nodes. Each media stream is partitioned into n equal size parts, called sub-frames, that are stored on the n different servers. First, the round trip delay between the client and each server is computed. Based on it, the starting time for each server is calculated and transmitted back to the servers. To guarantee the timely presentation of a single stream subject to jitter, for each sub-stream k, a total buffer b_k is provided

$$b_k = \lfloor 2 \times \Delta_k + \Delta_{max} - \Delta_{k+} \rfloor \quad \text{(II.3)}$$

where Δ_k is the jitter for substream k. Δ_{max} is the maximum jitter for all the substreams and Δ_{k+} is the maximum standard deviation of the propagation delay from the server to the client, for stream k. For each substream buffer, a lower water mark and an upper water mark are defined. When the buffer level falls outside of this range, then each server is notified to either skip some media units or pause.

Baqai *et al.* [6] from Purdue University propose five synchronization schemes for media units arriving from a server through a set of channels, assuming that the network uses a static reservation scheme and provides multiple channels with guaranteed bandwidth and delay bounds. All algorithms try to preschedule the transmission of the media units at the servers, so that they arrive at the destination before their play-out deadlines. Algorithm A makes a list of media units ordered by their play-out deadlines. The media units are then scheduled to be transmitted one by one on the earliest available channels. In algorithm B, media units are again scheduled in the order of their deadlines, and the scheduling time for transmission is computed such that the media unit is available at the client before its play-out deadline. Algorithm C also takes into consideration the size of the media units, favoring smaller size frames. Algorithm D forms the schedule as follows. Media units are scheduled for transmission according to their play-out deadlines. To account for the maximum jitter, the actual schedule is constructed by reducing all the schedule times by the maximum jitter. Algorithm E is identical to algorithm D, except that the initial list of media units is ordered by a combination of sizes and deadlines. Algorithms D and E are suited when destination buffers are severely limited and media units lost due to buffer overflow and deadline misses are tolerable. Algorithms B and C are most effective when the destination buffer is not severely limited and fewer deadline misses are desired. Algorithm A is most suitable when the destination buffer is not

a concern.

Little [29, 30] from Boston University, uses Petri nets for expressing temporal dependencies between streams. Each multimedia object has associated a start time and a duration. An object is associated with a stream and can contain one or more frames (for continuous streams) or one or more text/images. Based on this, a playout schedule for all streams can be computed and modeled by a Petri net. Each stream is also assumed to have a queue from which a frame is selected to be presented. Intrastream synchronization is done by controlling the queue level of each stream. If the queue level for the stream k is greater than nominal, frames are dropped. If it is lower than nominal, frames are duplicating. The workload variation is not taken into consideration.

These techniques are suited for creating multimedia presentations and would inquire overhead if used in live synchronization or record and playback of multimedia applications. In a live synchronization, they are not suited because the temporal relations between streams are not known in advance. If applied to record and playback of applications, then a program should convert the timing information between streams from one format (timestamps, synchronization events) into a Petri Net or another specific language format which adds unnecessary overhead.

II.5 Synchronization of the Shared Windows

To the best of our knowledge there is only one group at the University of Michigan, that studies the synchronization of audio, video and the shared windows stream. In this section we describe their results.

Mathur and Prakash [35] propose a protocol for synchronizing shared X windows and real-time audio in computer supported environments. They assume that the workstations have synchronized clocks. Since audio has stringent jitter and latency

requirements. audio is the master stream. while the windows stream is the slave. Audio packets arriving after their playback times are dropped. If a windows packet is received. it is put into a stack. When an audio packet arrives. it is played back along with the windows events from the stack that satisfy the condition $tw_{gen} \leq (ta_{beginrec} + 0.5ARECTIME)$. where tw_{gen} is the timestamp when the window event was received by the application at the sender. $ta_{beginrec}$ is the time the last played audio packet was recorded at the sender and $ARECTIME$ is the time it takes to record an audio packet. The protocol bounds how far the window-event stream can get ahead of the audio stream. It also adapts to situations where audio is ahead. by monitoring the asynchrony for a given number of window packets over a period of time. Asynchrony is defined as $ASYNC = (tw_{play_i} - ta_{play_j}) - (tw_{gen_i} - ta_{beginrec_j})$. where tw_{play_i} is the time the i^{th} window event is played. ta_{play_j} is the time the last audio packet j is played. tw_{gen_i} is the time the i^{th} window event was generated and $ta_{beginrec_j}$ is the time the last audio packet j began recorded. If the asynchrony is greater than a maximum value (100 ms). over a time interval longer than 500 ms and there are sufficient window events (more than 10). the protocol adapts by delaying the audio stream.

The protocol does not consider the effect of the load variation at the transmitter on the correctness of timestamps assignment. Also the use of synchronized clocks may restrict activities. Finally. it does not provide an extension for more than two streams.

Manohar and Prakash [33. 34] introduced the concept of replayable workspaces and propose a protocol that synchronizes time dependent and time independent (shared windows) streams. Synchronization uses a master/slave model. During the capture of the session. a synchronization event is posted at a well defined point of the master stream (e.g.. end of an audio frame) and is also inserted into all slave streams (e.g. the windows. video streams). During the replay of the session. the scheduling

of a synchronization event attempts to reset inter-stream asynchrony to zero. For any two streams (e.g., audio and windows), the synchronization algorithm proceeds as follows : if a window event is ahead of audio, it waits for matching audio frame. If this is a trend for the window stream (to be ahead), the algorithm compensates by decreasing the replay speed of window stream. If a window event is behind audio and this is a trend (to be behind), its replay speed is increased.

II.6 Synchronization in Distributed Systems

Son and Agarwal [3] from the University of Virginia present a synchronization model for recording and playback of distributed multimedia applications over ATM networks. The architecture suggested is the following. All workstations are connected to a multimedia server (*MMS*). When the session is recorded, every packet sent by a source is timestamped with the local time and sent to the *MMS*. In turn, *MMS* assigns to the packet a relative timestamps (*RTS*). At playback, synchronization is based on the relative timestamps. Frames that have the same *RTS* have to be played simultaneously.

RTS are assigned using a relation between the clocks of the source and the *MMS*. This relation is periodically determined, as follows. A session with very low jitter is established. A trigger packet is sent from the *MMS* to a site and after time t another trigger packet is sent. Upon receiving a trigger packet, the site timestamps it and sends it back to *MMS*. Let x_0, $x_0 + t$ be the instances at the source and y_0, $y_0 + t + w$ be the corresponding instances at the *MMS*. Then any instant x at a source will correspond to the *MMS* instant $y = ((t + w)/t)(x - x_0) + y_0$, with a maximum error $e = 2 \times maxrd(x - x_0)/t$, where *maxrd* is the maximum jitter from *MMS* to each source. After establishing the clocks offsets, the session is terminated.

At playback, when the destination receives a packet, it sends to the *MMS* the

time when the media data was displayed. Using the above relation, *MMS* normalizes the time and can detect with an error e if packets with the same *RTS* have been displayed at the same time. If z and z_1 are the times (according to the *MMS* clock) when two media units are displayed, then synchronization is guaranteed if $|z_1 - z| < |a| - |e|$, where a is the asynchrony and e is a threshold. The streams are out of synchrony if $|z_1 - z| > |a| + |e|$. In all the other cases, the synchronization between streams is not known. *MMS* adapts the rate of the slave streams, based on the detected asynchronies. The model may be extended also to sequential relations. For example, for a temporal relation A meets B, the timestamps for both the rear of A and the front of B are sent to *MMS*. For the relation A overlaps B, the timestamps of the rears of both A and B are sent to *MMS*.

This architecture can be also applied to real-time conferences, where data storage is not involved. Media data are first sent to a server (*SS*) which timestamps and sends them to the destination. The total error that may be introduced in detecting the asynchrony is $2e$. Since *SS* may become a bottleneck, more than one machine may be designated as *SS*. However, the streams that need to be synchronized have to use the same *SS*.

Rangan *et al.* [45] from the University of California at San Diego, address the problem of media storage and retrieval in a distributed system using a relative time system (*RTS*) kept by a server. Each media unit generated by a site is associated a *RTS*. The first media unit of the master stream starts the *RTS* and the successive units increment it. In order to associate a *RTS* to a slave stream unit, the server determines the *RTS* of the master media unit that is generated at about the same time as the slave media unit. If t_m and t_s are the arrival times of media units n_m (master stream unit) and n_s (slave stream unit) at the server, their earliest and latest possible generation times are $e_m(n_m) = t_m - M_{delay}$, $e_s(n_s) = t_s - M_{delay}$, $l_m(n_m) = t_m - m_{delay}$ and $l_s(n_s) = t_s - m_{delay}$, where M_{delay} and m_{delay} are the

maximum and minimum communication delays. Media units n_m and n_s have the same *RTS* if $max(l_m(n_m)-e_s(n_s), l_s(n_s)-e_m(n_m)) \leq E_{max}$, where E_{max} is a threshold value. Using the above relations, the server assigns a *RTS* to each media unit. The *RTS* is used later at playback. Every stream sends feedback units to the server. A feedback unit contains the *RTS* of the media unit that is currently scheduled for playback. Applying the above formulae to feedback units, the server detects which feedback units have been generated at the same time. Using this information, it finds the media units that are displayed simultaneously. Asynchrony at playback can be detected by comparing the *RTS* of a master media unit with the *RTS* of a slave media unit.

II.7 Motivation of Work

The temporal synchronization problem is a very important area of research in distributed multimedia systems. Consequently many solutions have been proposed in the last few years. Existing lip-synchronization solutions [4, 11, 16, 20, 43, 49] take into consideration the effect of the network, but they ignore the effect of workstation load on the synchronization specification and on the display time. The load on the sender machine may lead to an incorrect synchronization specification, which in turn may lead to an annoying presentation. The load on the destination workstation may determine variable display times of the media units which again may cause an annoying presentation. In this context, the main motivation of our lip-synchronization research is to address these problems. More precisely, our goal is to provide a lip-synchronization solution that dynamically adapts to both workstation and network load variations.

The solutions for synchronizing continuous and stateless discrete streams (other than the shared windows) [8, 9, 12, 13, 18, 24, 28, 45, 3, 60] also neglect

the effect of workstation load on the synchronization specification and on the display time. These algorithms cannot be directly applied to stateful discrete streams anyway, as they drop every discrete media unit that is late.

In Section 2.5 we have presented two solutions for synchronizing audio, video and the shared windows streams: one that addresses the synchronization between audio and the shared windows streams [35] and the other one which performs the synchronization of audio, video and the shared windows streams [33]. While the first solution delays audio when the shared windows stream tends to be behind, the second one changes the rate of the shared windows stream to catch up with the audio stream. From our experience, in a real-time video conference where the shared X client loads pages with heavy graphics, the shared windows stream is far behind the audio stream (6-7 seconds) due to the cumulative effect of large display times of the shared windows packets. In this situation, delaying audio as the first solution does, makes the presentation annoying. The second approach adapts the rate of sending shared windows packets to the X server. As the rate of playing these packets depends on the X server processing rate, this solution may also not work well for heavy windows updates.

The solutions existing so far [4, 8, 9, 11, 12, 13, 16, 18, 20, 24, 28, 43, 45, 49, 3, 60] ignore the issue of mixing audio streams while preserving the synchronization information. In this respect, they are limited to applications where only one user can speak at a time. In addition, all of them except [45, 3] consider only the case when the streams have a single origin, thus avoiding the issue of providing a common time for the application. Regarding this last issue, the solutions we have investigated, either have a link bottleneck [61], as the time is accessible through a modem connection to a mainframe, or have a workstation bottleneck, as they use a workstation to provide the time [45, 3, 19] which is a single point of failure. *NTP* [36], which assumes dedicated time servers that clients can access to adjust their times

creates a bottleneck in accessing the servers, too. As an alternative, we provide a scheme that assumes no dedicated time servers and no dedicated hardware.

Chapter III

Effect of Workstation Load

"A journey of thousand miles must begin
with a single step."

Lao-Tsu

A multimedia application has to be scheduled at regular intervals. At the source, this ensures a correct synchronization specification (no device driver queue overflow for continuous streams, and no delays in delivering shared windows packets to the application). At the destination, this ensures that the display time of media units is constant and that the playout deadlines of media units are satisfied.

A best-effort operating system cannot guarantee these times, as no operation bound is ensured by the time-shared scheduling policy. A straight solution is to use a preemptive operating system that gives to multimedia processes higher priorities than the rest of the processes running on the host. Some operating systems offer real-time extensions that satisfy these requirements (e.g., Solaris 2.5). In this chapter we present some experiments we performed in order to see if real-time is a solution for having a correct synchronization specification and a constant display time. If this was the case, then we could run the multimedia processes in real-time and our concern would be just the synchronization of the X-windows stream. As this was not the case, later in this chapter we introduce our model for a correct synchronization

specification and for estimating the display time of media units.

III.1 Exploring Real-time Capabilities

The real-time capabilities of currently used operating systems allow a user to specify the scheduling class of a particular process. This by default is time-sharing class. If real-time class is used, the process is given a high priority which may be even higher than the priority of system processes. Under this condition, one would expect that by scheduling audio and video in real-time, their stringent time requirements will be satisfied.

Next we present some experiments we performed to see if the real-time extensions of the currently used operating systems can guarantee the deadlines of multimedia processes.

III.1.1 Experimental Design

Using the experimental setup described in Section I.3, we have tested both scheduling policies for multimedia processes: time-sharing and real-time.

The audio process was initialized with the following parameters: 8KHz sampling rate, 8 bit precision, mono channel and μ-law encoding. The video board was initialized with a skip factor of 2, which results in 10 frames/sec rate (at the application).

We measured the time difference between two consecutive reads from the audio device (ideally this should be 125ms) and two consecutive captured frames from video board (ideally, this should be 100 ms). We call these times the audio and video *inter-arrival* times, respectively.

In order to investigate how other processes influence the inter-arrival time for audio and video, while running the audio/video process we run typical activities for

Table III.1: Variation of the inter-arrival time [ms].

Audio		*Video*	
Concurrent activity	Standard deviation	Concurrent activity	Standard deviation
none	0.744	none	3.853
read from disk	1.094	read from disk	4.334
print on the console	3.636	print on the console	7.179
20 busy processes	55.914	20 busy processes	78.623
random memory write	8.950	random memory write	10.390
Mosaic	54.96	Mosaic	72.509
video	9.688	audio	4.990
video, read from disk	12.065	audio, read from disk	8.892
video, print on the console	15.068	audio, print on the console	14.340
video, 20 busy processes	93.952	audio, 20 busy processes	113.187
video, random memory write	14.590	audio, random memory write	12.131
video, Mosaic	107.085	audio, Mosaic	122.295

Table III.2: Effect of real-time scheduling.

Audio (RT)		*Video (RT)*	
Concurrent activity	Standard deviation (msec)	Concurrent activity	Standard deviation (msec)
20 busy processes	0.093	20 busy processes	0.117
video(RT), Mosaic	6.046	audio(RT), Mosaic	7.057

a workstation usage :

- I/O bound - a process repeatedly reads a 3 Mbytes file from a server disk. In another experiment, a process just prints at the console
- CPU computation - a process initializes a variable in an infinite loop. To see the effect of increasing CPU workload we run one, two up to twenty copies of this process.
- memory bound - a process randomly writes in a 1000×1000 matrix to simulate page faults.
- interaction with X Server - run Mosaic* and move windows on the screen while loading pixmaps.

III.1.2 Measurements

Table III.1 shows the standard deviation of the audio and video inter-arrival times in the presence of the corresponding load. Figure III.1 shows the variation of the video inter-arrival time in each of the experiments. The graphics for audio experiments show a similar behavior, so we do not present them here. Moreover, since the video process requires more time to process a frame than the audio process needs for reading audio data, after each run, the priority of the video process decreases with a greater value than the audio process priority and hence the inter-arrival time for video shows larger variations than the inter-arrival time for audio [62].

We have repeated the experiments using the real-time scheduling capabilities of Solaris. As expected, the results improved, so we show here only the values for the experiments where the behavior in non-real time was worst (running twenty busy

*At the time we conducted these experiments, Netscape was not widely available.

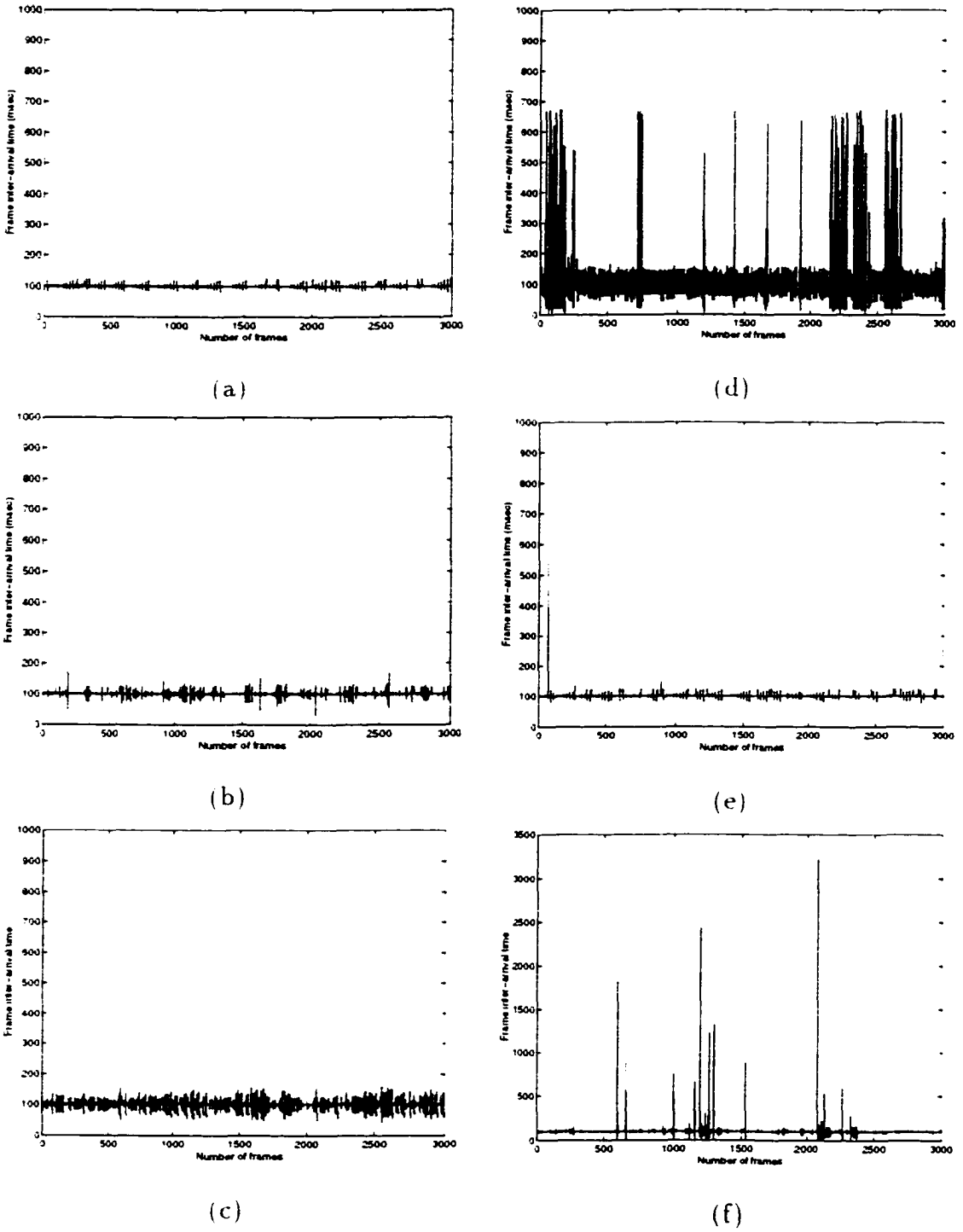
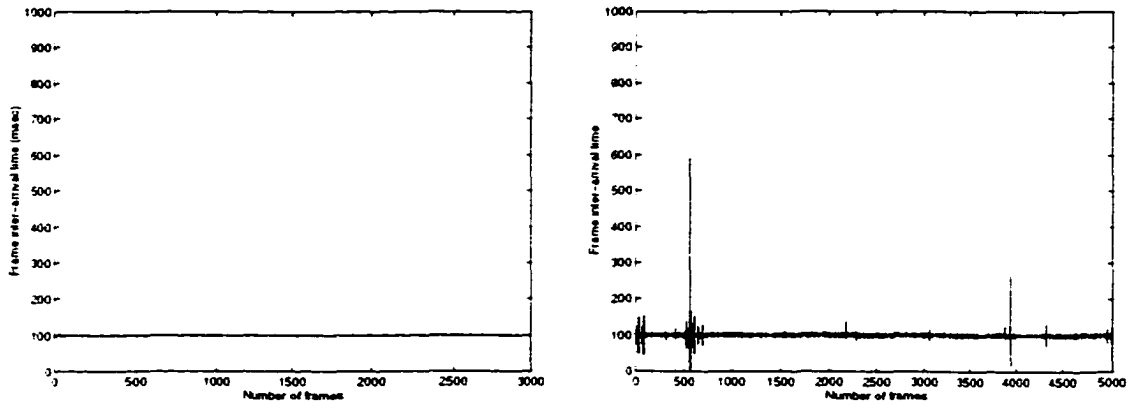


Figure III.1: The video inter-arrival time variation when video, audio and the following job was running: (a) none, (b) read from disk, (c) print on the console, (d) twenty busy processes, (e) random memory write and (f) Mosaic, move windows on the screen.



(a) (b)

Figure III.2: The video inter-arrival time variation in real time when the following job was running : (a) twenty busy processes (b) Mosaic, move windows on the screen.

processes and running Mosaic). Table III.2 shows the standard deviation in each case. Figure III.2 shows the variation of video inter-arrival time.

III.1.3 Results Interpretation

From these experiments we see that both audio and video are most influenced when twenty busy processes were running or when we run Mosaic and move windows on the screen. Even though the standard deviation is small in all of the experiments, and one might conclude that on the average, the behavior is very good, this is a result of a mix of very small and large $inter - arrival$ times. If the $inter - arrival$ time is greater than the time required to fill the kernel audio/video device drivers queues, this will result in an overflow and data losses. This fact has to be taken into account by the synchronization specification, since it directly affects the interstream synchronization.

When running video and Mosaic, the highest spike in the video inter-arrival time was 3 seconds. Similar values were obtained for audio. Twenty busy processes

introduce many spikes around 0.7 seconds. When running video, audio and Mosaic, the highest spike for video inter-arrival time was 3.3 seconds. Again, spikes around 0.7 seconds appear when twenty busy processes run. As expected, the greater the number of busy processes, the more the performance of the video/audio processes degrades.

Although busy processes affect audio and video, the worst inter-arrival time variation for both audio and video was obtained when Mosaic was running and windows were moved on the screen. Since Mosaic involves not only interaction with the window system but also communication, we wanted to isolate the effect of communication. To do so, while running the video process we run a process that was continuously executing "ftp" from a remote site. In this experiment, the variations were small. In another experiment, we run Mosaic to load pixmaps and move almost all the time the windows on the screen so as to emulate high interaction with the X Server. In this experiment we obtained high variations. Therefore, we conclude that the large variation of the inter-arrival time of video when Mosaic is run is due to the interaction with the windows system which sometimes consumes too much time and deprives the video process to be scheduled at the required intervals.

The other remaining experiments showed very small inter-arrival time variation. Random memory operations introduce variations only at the beginning, when pages are loaded into memory (compulsory misses). Reading from disk does not have much influence on multimedia performance, because a buffer is allocated at the beginning and since the program just reads from disk into this buffer, no other page faults occur. Printing on the console has negligible influence for both audio and video.

Real time eliminates the inter-arrival time variation in case of twenty busy processes, but not in the case of Mosaic and windows movements. This is because the X windows system is not fully-preemptive and thus the deadlines of real-time processes may be missed.

We did not perform any experiments with the data sharing process, as X requests are generated in bursts, so XTV does not need to be scheduled at regular intervals.

From the experiments we presented, we see that the real-time scheduling class is not always capable of ensuring the time constraints associated with the audio and video processes. This is the reason why in our work we study the temporal synchronization problem in best-effort systems.

III.2 Media Synchronization Specification

Ideally, the existing temporal relations between media units when the streams are captured, are exactly preserved when they are played. Unfortunately, due to the best-effort nature of the current networks and operating systems, achieving this goal is challenging. Media units arrive at the source application at various times, and thus the synchronization specification assigned by the application may be different than the real temporal relation between the media units when they are captured. As the destination application uses the synchronization specification to present the streams, a wrong synchronization specification triggers an erroneous presentation. To better understand the requirement for a correct synchronization specification, we give a brief overview of the functionality of multimedia devices and the mechanism of sharing X-windows used in our research.

III.2.1 Acquisition of Continuous Streams

Continuous streams, audio and video are captured by audio and video devices. The two basic functions of an audio device (e.g., Sun Audio) is to record and play audio data. To minimize delays, whenever the device driver has accumulated a buffer of data (the size of the buffer can be defined by the user), it takes the data and puts

it into a kernel queue. When the audio process is scheduled, it reads one buffer from the kernel queue. If the queue is full, the audio driver will no longer put data into the queue. Next recorded audio is lost until the application reads data from the kernel queue. Note that even if the application flushes the kernel queue at every read, overflow may still happen if the time between two consecutive scheduling intervals of the application is larger than the time it takes the audio driver to fill the kernel queue.

A video device (e.g., SunVideo) can capture a maximum of 30 frames/sec. However, the application can program the video device to provide frames at a smaller rate, by specifying a *skip factor*. In this case, the video device still captures 30 frames/sec, but compresses and stores in a local queue, every *skip factor* + *1* frame. For example, if the skip factor is 0, it stores every frame (the rate is 30 frames/sec), whereas if the skip factor is 2, every third frame will be compressed and stored in the queue (the rate is 10 frames/sec). When the video process is scheduled, it gets one frame from the video board queue. If the queue is full, the video device overwrites the oldest frames. Even if the video process flushes the queue every time, the queue may overflow if the time between two consecutive scheduling intervals of the video process is larger than the time to fill the queue. Note that the smaller the queue size, the smaller the latency [59], and the larger the queue, the smaller the number of frames lost. The optimum size of the queue is 2-4 buffers [59] and in this case the queue is filled in 400 ms (for 10 frames/sec frame rate).

Note that basically, the data acquisition of audio and video devices is the same, with one difference. When the video queue overflows, old frames are lost, while in case of audio queue overflow, new data are lost.

III.2.2 The Mechanism of Sharing X-Windows

In our thesis we use *XTV* [2] as the mechanism to create a shared workspace on top

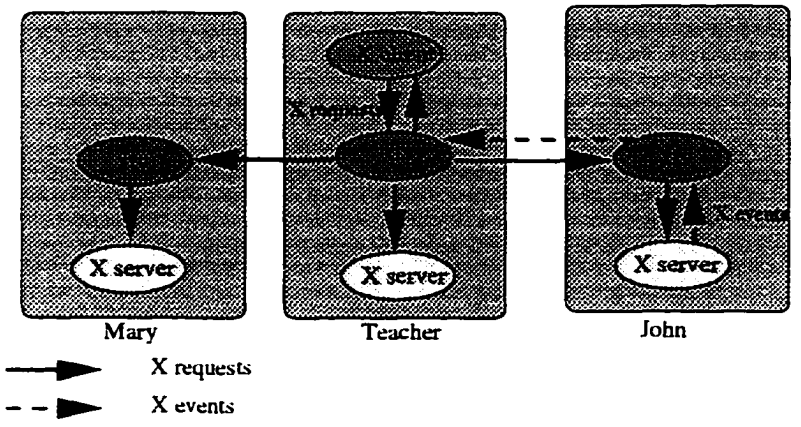


Figure III.3: The mechanism of sharing X clients using *XTV*.

of X windows. *XTV* runs on every host the videoconference application runs (see Figure III.3). An X client runs only on one host. Once the X client (e.g. Netscape) is started, *XTV* captures the output of the X client (shared windows packets or X requests [46]) and sends it to the local X server and to the remote *XTV* processes. A remote *XTV* process receives the X client output and sends it to the local X server. At one moment only one user can interact with the X client. His interaction (X events [46]) is sent to the *XTV* process where the X client runs. This *XTV* process sends these X events to the X client.

It is worth mentioning that the output of the X client is not sent immediately to the X server, but it is buffered by *Xlib*, a layer that implements the X protocol [46]. This is done in order to minimize the waiting time to gain access to the network. Also, the X server implements a round-robin policy in serving its X clients, so requests coming from X clients are queued and served only when the X client is scheduled.

Also, the shared windows stream has a history (e.g., a request to create a window is related to the previous request which creates the parent window) and thus, if audio and video media units can be dropped in order to keep the streams

synchronized, an X request can only be selectively dropped (e.g., an X request asking the X Server to draw a line may be dropped).

III.2.3 Specification for Continuous Streams

Our synchronization specification model uses *numerical timestamps* (frames sequence numbers). Our goal is to assign to each frame its correct sequence number with respect to the order in which it is captured by the device driver, and not to the order in which it is delivered to the application. As we have shown in the previous section, the lost frames (due to the device driver buffer overflow) introduce gaps in the sequence numbers of the frames delivered to the application. In this section, we show how these sequence numbers can be actually computed.

The frame sequence number depends on the policy implemented by the device driver when its queue overflows. Further we consider two of the most common policies: (1) the device driver overwrites the old frames (in a circular fashion), and (2) the device driver discards the new frames. An example of a device driver that implements the first policy is the Sun video device, while an example of a device driver that implements the second policy is the Sun audio device. Next, we show how these two policies affect the frame timestamping.

In both cases we make the following two assumptions: (1) no buffer overflow occurs before the process reads the queue for the first time, and (2) once the process is scheduled, it reads all the buffers from the queue†. For a stream α, we denote by $lost_\alpha$ the number of frames lost while the process waits to be scheduled. Let $diff_\alpha$ be the time difference between the last two read operations, let n_α be the number of buffers of the device driver, and let d_α be the frame duration of stream α. For all

†In our implementation, we try to enforce the first assumption by reading data from the queue immediately after the device driver is opened. To enforce the second assumption, we use a special thread to read the buffers from the queue and deliver them to the application.

Table III.3: Notations.

α_i	the sequence number of the $i-th$ frame received by the application
α_{play}	the sequence number of the frame of stream α that is currently playing
d_α	duration of a frame of the stream α
n_α	number of buffers in the device driver queue of stream α
$diff_\alpha$	the time difference between the last two consecutive read operations α
$lost_\alpha$	number of frames of stream α that are lost between the last two read operations, due to device driver queue overflow
t_{0_α}	starting time for stream α
$A_{\alpha\beta}$	maximum acceptable asynchrony between streams α and β
$t_{\alpha\beta}$	tolerance (maximum acceptable asynchrony between streams α and β expressed in number of frames)

these notations see Table III.3. Then the number of frames which are lost is:

$$lost_a = \begin{cases} \left\lceil \frac{diff_a - n_a \times d_a}{d_a} \right\rceil & \text{if } diff_a - n_a \times d_a > 0 \\ 0 & \text{otherwise} \end{cases} \quad \text{(III.1)}$$

Our solution for assigning a frame sequence number (α_c) is based on the device queue type. Type1 queue is when the device driver overwrites the oldest frames (e.g. Sun video device driver) and type2 queue is when the device driver no longer puts data into a full queue (e.g. Sun audio device driver). With these considerations, our algorithm of assigning sequence numbers is as follows :

```
when process is scheduled {
    getFrame(fr): /* read frame from queue */
    if (Type1Queue)
      α_c = α_c + 1 + lost_a: /* compute next sequence number */
    stampFrame(fr, α_c): /* assign seq. num. to current frame */
    while (queue ≠ ∅) {
        getFrame(fr):
        α_c = α_c + 1:
        stampFrame(fr, α_c):
    }
    if (Type2Queue)
      α_c = α_c + 1 + lost_a: /* compute next sequence number */
}
```

As an illustration, consider the following example. Assume a Type1 queue with three buffers (i.e., $n_a = 3$), and that at time t_0, when the process is scheduled for the first time, the queue contains exactly two frames: 1 and 2. Then, after the process reads both frames (assigning to them the sequence numbers $\alpha_1 = 1$, and

$\alpha_2 = 2$, respectively), assume that the next time when the process is scheduled is $t_1 = t_0 + diff_a$, where $diff_a = 5 \times d_a$. Since between t_0 and t_1, the device driver has written five frames in the queue (i.e., frames 3, 4, 5, 6, and 7, respectively), and since the queue has only three buffers, the content of the queue at time t_1 will be 5, 6, and 7. When the process reads the first frame at time t_1, then it assigns the timestamp $\alpha_3 = \alpha_2 + 1 + lost_a$, where $lost_a = \lceil \frac{diff_a - n_a \times d_a}{d_a} \rceil = 2$, which finally gives us the correct value $\alpha_3 = 2 + 1 + 2 = 5$. Following the next two frames will receive the sequence numbers $\alpha_4 = \alpha_3 + 1 = 6$, and $\alpha_5 = \alpha_4 + 1 = 7$, respectively.

As an example for a Type2 queue, consider again a queue with three buffers (i.e. $n_a = 3$). Similarly to the previous example, assume that at time t_0, when the process is scheduled for the first time the queue contains two frames: 1 and 2. Thus, according to the algorithm, the sequence numbers assigned to these frames will be $\alpha_1 = 1$, and $\alpha_2 = 2$, respectively. Next, assume that the next time t_1 when the process is scheduled is again after $diff_a = 5 \times d_a$. However, since in this case, when the queue is full the new frames are lost, the content of the queue at time t_1 will be 3, 4, and 5. Then, when the process reads all the frames from the queue at time t_1, it assigns the sequence numbers $\alpha_3 = 3$, $\alpha_4 = 4$, and $\alpha_5 = 5$, respectively. Moreover, after the buffer is empty, the process computes the sequence number for the first frame that will be read *next* time, i.e., $\alpha_6 = \alpha_5 + 1 + \lceil \frac{diff_a - n_a \times d_a}{d_a} \rceil = 5 + 2 + 1 = 8$. Note that this is the correct sequence number since frames 6 and 7 have already been lost (due to the buffer overflow).

III.2.4 Specification for the Shared Windows Stream

In assigning correct sequence numbers to audio and video media units, we took advantage of the fact that the streams are periodic. On the other hand, the X windows stream is *aperiodic* and the X requests do not contain any time information. As a result we cannot apply the same procedure for computing the correct timestamps

in the X windows case.

The timestamp of an X request is the moment of time the X request has been generated by the X client. Our goal is to estimate this time. Let $T_{Xclient}$ be the time when the request is initiated by the X client, T_{app} be the time when the request is received by XTV (the data sharing process), and $Prop_{Xclient->app}$ be the time interval needed to deliver the request from the X client to XTV. Thus, we have:

$$T_{Xclient} = T_{app} - Prop_{Xclient->app} \quad \text{(III.2)}$$

T_{app} can be simply computed by calling *gettimeofday* when XTV receives the X request. To estimate $Prop_{Xclient->app}$ we have implemented a producer-consumer application based on UNIX sockets, as they are used to communicate between the X client and XTV on the same machine.

The producer sends variable size packets (power of 2) to the consumer. Whenever the consumer receives a packet, it sends the packet back to the producer. Table III.4 shows the total time elapsed (RTT) from the moment the producer has sent a packet until it receives the packet back (note that here $RTT = 2 \times Prop_{Xclient->app}$). We have repeated the experiment in the presence of various loads, by running concurrently up to three busy processes.

For packets smaller than 8192 bytes, the RTT time is less than 1 ms, i.e., $Prop_{Xclient->app}$ is less than 0.5 ms. As expected, for larger packet sizes, the RTT increases both with the packet size and with the load introduced in the system. Since excepting *PutImage*, all the other X requests consist of several bytes, we neglect $Prop_{Xclient->app}$. In the case of *PutImage*, using the experimental data in Table III.4 we estimate $Prop_{Xclient->app}$ based on the image size (packet size) and we assume that there is one busy process in the system (corresponding to the first column in the table). This choice is supported by our experiments in which we have found that the activity generated by the IRI processes is approximately equivalent

to the activity generated by one busy process.

Though using *gettimeofday* in estimating T_{app} introduces certain measurement errors, and estimating $Prop_{Xclient->app}$ for *PutImage* is not very accurate, in practice computing $T_{Xclient}$ based on these values works reasonable well. One of the main reasons for this is that the accepted asynchrony between X windows and audio is within the range (-500, +750) ms [54], i.e., one order of magnitude larger than the accepted asynchrony between audio and video (+/- 80) ms.

For uniformity we use a sequence number to stamp the X request, instead of time. The sequence number is computed as the sequence number of the audio frame that was captured when the X request was initiated by the X client. If there is no such audio frame, the X request is stamped with the sequence number of the corresponding video frame. If no video stream is captured, then the sequence number of the X request is -1, meaning that X windows will not be synchronized at the destination with any stream.

III.3 Media Display Time

When a media unit arrives at the destination application, it is sent to the presentation device according to the timings specified by the synchronization specification. However, the user sees the effect of playing the media unit only at the end of its display time. Usually the display time of audio frames is very short (negligible), but the display time of video frames is large (e.g., an average of 243 ms for a 24 bits depth, 640 × 480 pixels windows) and even larger for some X-windows media units (e.g., 475 ms to put an image in Netscape). Moreover, due to workstation load variation, even for the same media unit, the display time may vary. In this situation, we need an estimation of the display time for each type of media unit so that the destination knows when to send each media unit to its presentation device.

Table III.4: The RTT time for a Unix socket in the presence of various loads.

Packet size [bytes]	RTT [ms] (no load)	RTT [ms] 1 busy process	RTT [ms] 2 busy processes	RTT [ms] 3 busy processes
2	0.15	0.17	0.17	0.17
4	0.15	0.17	0.17	0.17
8	0.15	0.17	0.17	0.18
16	0.15	0.18	0.18	0.18
32	0.16	0.18	0.18	0.18
64	0.16	0.18	0.18	0.18
128	0.16	0.19	0.19	0.18
256	0.17	0.19	0.19	0.19
512	0.19	0.21	0.21	0.21
1024	0.24	0.24	0.25	0.25
2048	0.26	0.27	0.27	0.29
4096	0.39	0.38	0.39	0.34
8192	0.68	0.57	0.58	0.58
16384	2.4	2.8	11.81	21.26
32768	4.46	6.05	27.16	27.16
65536	8.75	10.20	57.19	149.65
131072	22.39	55.61	128.52	265.65
262144	46.43	146.62	278.45	307.25

The display time of a media unit is given by the following relation

$$DisplayTime = Prop_{app->presDev} + ProcessTime_{presDev} \quad \text{(III.3)}$$

where $Prop_{app->presDev}$ is the time it takes to send the media unit from the application to the corresponding device and $ProcessTime_{presDev}$ is the time it takes the device to process the media unit. Video frames and X-windows requests are sent to the X server (as presentation device) via Unix socket connections. An audio packet is sent to the audio device (as presentation device) by copying the audio frame to a system buffer.

III.3.1 Estimation for Continuous Streams

To estimate the display time for the video stream, first, on the testbed described in Section I.3 , we have conducted experiments to see how various jobs influence the display time. Video frames (320 × 240 pixels) were CellB [59] hardware compressed, software decompressed and displayed in a 24 bits depth window. We measured the display time as the time difference between the moment the video process calls the display function (*XShmPutImage*) until the X Server sends back the event meaning that the display function has completed (*ShmCompletion*). We also measured the total processing time of a video frame (which includes both the decompressing and display times). Concurrently with the video process we run typical activities for a workstation usage :

- *none* - the video process runs alone on an idle workstation.
- *X server bound* - the application window is moved while the video stream is displayed. This puts additional load on the X server process which may delay the display of the frame in order to repaint other portions of the screen.

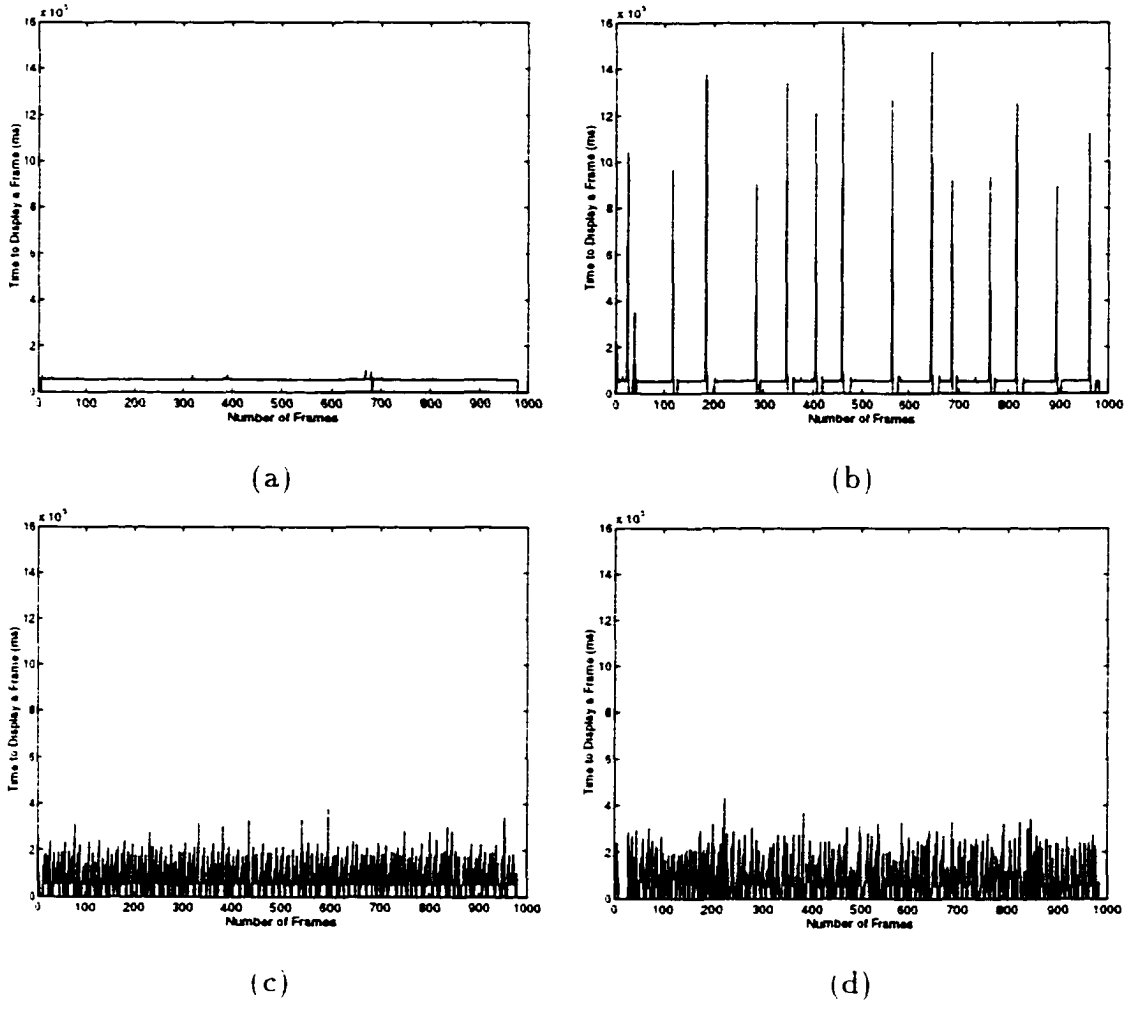


Figure III.4: Effect of load on the display time of a video frame when: (a) no other load was introduced in the system, (b) the window was sometimes moved, (c) a busy process was concurrently running, and (d) another video image was displayed.

- *CPU bound* - besides the video process, we run a simple computation bounded process (that initializes a variable in an infinite loop).

- *moderately increase both CPU and X server activity* - two video streams are concurrently displayed.

We measured the display time and the total processing time for 1000 frames. Experiments showed that on the average 84.28% of the total processing time was spend by displaying the frame and only 15.72% of the time was spend on decompressing the frame. Since the curve of the variation of the total processing time and the curve of the variation of the display time are close, we show here only the variation of the display time. Figure III.4 shows the variation of the display time of a video frame in the each of the experiments mentioned above. When no additional load was put on the system, the average time to software decompress a frame was 8ms, while the average time to display a frame was 46 ms. In the presence of another process (video or busy process), the display time average almost doubles (83 ms).

The display time of a 24 bits depth, 640×480 pixels window follows the same variation, with an average of 243 ms. If an audio and video frames are to be played at the same time, and they are sent at the same time to their presentation devices, assuming audio plays immediately (like [43]), there is 243 ms skew between the frames when they are visible to the user. As the desired skew range is $(-80, 80)$ ms, and the acceptable skew range is $(-160, 160)$ ms, the two frames are actually completely out of sync at the end of the video display time. This is the reason why in the case of large size windows (640×480) and even for medium size windows (320×240 pixels) in the presence of workstation load, the application needs to estimate the video display time in order to keep the streams synchronized.

After collecting the data, we processed them off-line. Basically, we take the total time to process a video frame to be the time interval from the moment the

video frame was sent to the X server until an acknowledgement is received from the X server:

$$TotalTime = Prop_{app->Xserver} + ProcessTime_{XShmPutImageReq} + Prop_{Xserver->app} \quad \text{(III.4)}$$

where $Prop_{app->Xserver}$ is the propagation delay from the video process to the X server, $ProcessTime_{XShmPutImageReq}$ is the processing time of the X request to display the frame, and $Prop_{Xserver->app}$ is the time it takes to send the acknowledgment from the X server to the application. As the acknowledgment is a 32 bits packet, in which case the propagation delay is around 0.8 ms, we will ignore it (see Table III.4: as the RTT for 32 bits packets is 0.16 ms, the propagation time in one direction is 0.8 ms). With this consideration, the display time of the video frame is

$$DisplayTime = Prop_{app->Xserver} + ProcessTime_{XShmPutImageReq} \quad \text{(III.5)}$$

and it equals our measured time ($TotalTime$).

To estimate this time we use exponential averaging:

$$E_k = cM_{k-1} + (1-c)E_{k-1} \quad \text{(III.6)}$$

where E_k is the estimated display time of frame k, while M_{k-1} is the measured display time of frame k -1.

The criteria we used in determining c was to minimize the standard deviation. For this we varied c in the range [0.05, 0.95] in steps of 0.05. We observed that for all the experiments the estimated value of the display time depends mostly on the previous estimated value. When the additional job was to move windows, this is mainly because moving a window generates spikes which have practically no impact

on the display times of the frames once the movement stops. For the other additional jobs, the explanation of this behavior is that increasing the CPU and the X server load and keeping it constant for some time increases the value of the display time. This results in relatively small variations between the old estimated display time and the currently measured display time, which makes the computation of the new estimated display time to be little influenced by the value of c over a large range[‡] However, to account for the case when windows are moved on the screen, we give a higher weight to the old estimated time. In our experiments, the standard deviation was minimized when c was between 0.2 and 0.3. Therefore, in our implementation we choose $c = 0.25$.

Finally, in estimating the display time of the audio frame we make the same assumption as Elefteriadis [16] and Nahrstedt [43], i.e., we assume that the audio stream plays continuously. To estimate the display time of frame a_i, we query the audio device for the sequence number of the currently playing audio frame, a_{play}. Then a_i will play after a time interval equal to $(a_{play} - a_i)/timesd_a$, where d_a is the media unit duration of the audio stream.

III.3.2 Estimation for the Shared Windows Stream

While for estimating the display time of a video frame we used the acknowledgement generated by the X server, we cannot rely on this mechanism to estimate the processing time of all X requests. This is simply because not all X requests generate acknowledgments.

We address this problem by sending a probe request (*GetKeyboardMapping* [46]) that forces an acknowledgment after each such an X request. When we get the reply back, since the X server processes the requests in a first-come first-served order, we

[‡]Note that, at the limit, when the old estimated value is equal to the current measured value, the new estimated value is independent of c.

know that the request of our interest has also been processed. We measured the total time elapsed from the moment we sent the request to the server until we receive back the reply corresponding to *GetKeyboardMapping* request. The total measured time can be divided as follows:

$$TotalTime_{Xrequest} = TimeAtXlibLayer + Prop_{Xlib->Xserver} + ProcessTime_{Xreq} + ProcessTime_{probeReq} + Prop_{Xserver->app} \quad \text{(III.7)}$$

where $TimeAtXlibLayer$ is the time spent by the X request at the Xlib layer [46], $Prop_{Xlib->Xserver}$ is the time it takes the X request to be delivered to the X server, and $Prop_{Xserver->app}$ is the time it takes to send the reply back to the application. In addition, $ProcessTime_{Xreq}$ and $ProcessTime_{probeReq}$ are the times the X server processes the X request and respectively the probe request.

If only the probe request were sent to the X server, then the total time measured from the moment the probe request has been sent to the X server until its reply is received by the application is given by

$$TotalTime_{probeReq} = Prop_{Xlib->Xserver} + ProcessTime_{probeRequest} + Prop_{Xserver->app} \quad \text{(III.8)}$$

Substituting the right hand side of (III.8) into (III.7), we obtain

$$TotalTime_{Xrequest} = TimeAtXlibLayer + ProcessTime_{Xreq} + TotalTime_{probeReq} \quad \text{(III.9)}$$

The display time of an X request is

$$DisplayTime = Prop_{Xlib->Xserver} + ProcessTime_{Xreq} \quad \text{(III.10)}$$

Substituting $ProcessTime_{Xreq}$ from (III.9) into (III.10) we have

$$DisplayTime = Prop_{Xlib->Xserver} + TotalTime_{Xreq} - TimeAtXlibLayer - TotalTime_{probeReq} \quad \text{(III.11)}$$

Since the probe request causes the Xlib to send *immediately* all previous requests to the X server, we will neglect $TimeAtXlibLayer$. In addition, since the majority of the X requests have less than 32 bits, we will also neglect $Prop_{Xlib->Xserver}$. (Recall from Section III.2 that this time is less than 0.8 ms.) However, for the *PutImage* request where a packet can have a large size, we estimate the propagation time using Table III.4, first column.

With these considerations, the relation to compute the display time when we neglect the $Prop_{Xlib->Xserver}$ time is:

$$DisplayTime = TotalTime_{Xreq} - TotalTime_{probeReq} \quad \text{(III.12)}$$

and the relation to compute the display time for *PutImage* request is:

$$DisplayTime = = Prop_{Xlib->Xserver} + TotalTime_{Xreq} - TotalTime_{probeReq} \quad \text{(III.13)}$$

To measure the display time of an Xrequest, we first measured off-line the total time it takes to send the probe request to the X server and to receive back the reply from the X server ($TotalTime_{probeRequest}$). Whenever we send an X request that does not ask for a reply, we also send a probe request. Then we measure the total

time it takes from the moment the X request has been sent to the X server until the reply for the probe request is received back, i.e., ($TotalTime_{Xrequest}$). The display time of the X request is computed then as the difference between $TotalTime_{Xrequest}$ and $TotalTime_{probeRequest}$. In the case of the *PutImage* request, we add the $Prop_{Xlib->Xserver}$ time to this difference.

In our measurements we found that the most expensive requests are the ones which result in window creation and updating (a complete list is given in Appendix A). For example, *CreateWindow* takes around 220 ms, and *ConfigureWindow* takes around 175 ms. Some of these requests, such as *PutImage*, are highly variable, as they depend on their content. For example, it takes only 13 ms to load the maximize/minimize/close icon, while it takes up to 475 ms to load a 3 square inches image in Netscape. Similarly, the *PolyFillRectangle* request takes 73 ms to fill the xterm's scroll bar, while it takes 210 ms to fill a 2 square inches rectangle with a special pattern.

The next most expensive requests are queries (requests that ask for a reply from the X server) like *QueryColors*, which take on the average 47 ms. Following are the requests that create resources other than windows (e.g., *CreatePixmap*) which take between 10 ms and 50 ms. The remaining requests, such as the ones that destroy resources (e.g., *FreePixmap*), change resource properties (e.g., *ChangePointerControl*), and map/unmap windows (e.g.,*MapWindow*) take less than 15 ms.

III.4 Summary

Current distributed multimedia applications are mostly designed, implemented and used on top of general-purpose operating systems (e.g. UNIX) and Internet protocol stacks. Within this best-effort environment, to achieve user acceptance for a synchronized presentation, the distributed application must balance the nondetermin-

istic behavior of the underlying operating system and network. From the temporal synchronization point of view, this may cause two things. The first one is that the synchronization specification assigned by the source application may be wrong. This is because media units may not arrive at the same time to the source application. The second one is that the display time of media units may vary. This is because the process that displays the media unit has to compete with other processes for a CPU share. Under these conditions, two media units with considerable different display times may be out of sync, even if they both have been sent at the same time to their presentation devices.

Traditional existing solutions ignore the effect of workstation load on the temporal synchronization and focus only on the effect of network. To address this problem, we first study if the real-time capabilities of existing general purpose operating systems can schedule multimedia processes at regular intervals. Practically we have shown that although in many situation this is the case, in the case of high X windows interaction, the processes are scheduled again at irregular intervals, because X windows is not fully preemptive.

As real-time does not eliminate the time variability in scheduling multimedia processes, we have presented our mechanism that provides a correct synchronization specification. Also, we have provided extensive analysis of the display time of media units and we have described suitable solutions for estimating the display time for each type of stream.

Chapter IV

Synchronization Algorithms

"Discovery consists in seeing what everyone else has
seen and thinking what no one else has thought."

Albert Szent-Gyorki

To achieve a continuous presentation under a time-sharing multiprocessing operating system, the synchronization quality of traditional synchronization mechanisms may vary according to the workload of the system. When the system encounters an overload situation, the synchronization usually fails. In order to achieve our objective of synchronizing audio, video and shared windows we first introduce in Section IV.1 our synchronization condition. Next, Section IV.2 describes our lip-synchronization algorithms, and section IV.3 describes our algorithms for synchronizing audio, video and the shared windows streams.

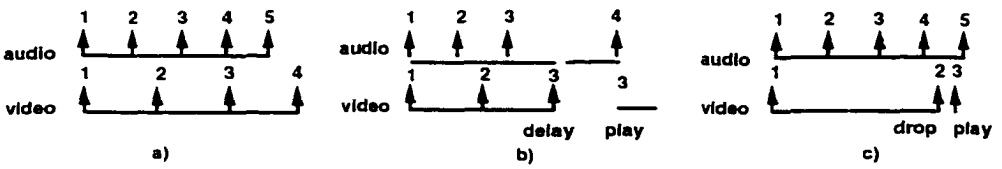


Figure IV.1: Intuitive interpretation of the model (a) ideal case, (b) when video is ahead, (c) when video is late.

IV.1 Synchronization Condition Between Streams

Usually, any synchronization algorithm defines certain conditions that streams should meet in order to be synchronized. Examples of such synchronization conditions are: (1) frames with the same sequence number should play simultaneously [11], (2) the difference between the acquisition timestamps of the master and the slave frames should be smaller than the accepted asynchrony between the streams [12, 13, 16, 24, 43, 45, 49, 3], and (3) streams should all reach a synchronization point in order to play [33]. Let us assume that the synchronization specification assigns correct sequence numbers or timestamps to the frames, as explained in the previous chapter. Then, the first and the third conditions restrict the streams to have either the same frame duration (first condition), or the frame durations to have a common divisor (third condition). On the other hand, the second condition requires timestamps to be used for the synchronization specification, which may waste valuable network bandwidth. Moreover, this information may be redundant, since the frames need to have anyway sequence numbers in order to detect losses, if the transport protocol does not provide reliability (e.g., UDP). For these reasons, our objective is to define a synchronization condition based on sequence numbers, and which can handle streams with arbitrarily frame durations.

Consider two streams, one is the master, the other is the slave [54]. Our objective is to find the sequence number of the frame of the master stream that should play if a certain frame of the slave stream would start. The utility of our model is intuitive. Figure IV.1, shows the case when one audio stream and one video stream have to be synchronized. Audio is the master stream. Whenever a video frame is to be displayed, we compute the sequence number of the audio frame that should ideally play if this video frame would start. If the sequence number of the currently playing audio frame, matches the computed value, then the frame plays immediately

(Figure IV.1(a). If the playing audio frame has a smaller sequence number, then the video frame waits (Figure IV.1(b)). If the playing audio frame has a larger sequence number, then the video frame is dropped.

Next, we compute the sequence numbers (α_i) of the frames of the master stream that should play while a frame (β_j) of the slave stream plays. Note that there may be more than one frame of the master stream that plays while frame β_j plays, but it can be only one frame α_i that plays when frame β_j starts. Then, the following relations hold (see Table III.3 for notations):

$$\alpha_i = \left\lfloor \frac{t - t_{0_\alpha}}{d_\alpha} \right\rfloor . \tag{IV.1}$$

$$\beta_j = \left\lfloor \frac{t - t_{0_\beta}}{d_\beta} \right\rfloor \tag{IV.2}$$

Replacing time t from (IV.2) in relation (IV.1), we have:

$$\beta_j \frac{d_\beta}{d_\alpha} + \frac{t_{0_\beta} - t_{0_\alpha}}{d_\alpha} - 1 < \alpha_i < \beta_j \frac{d_\beta}{d_\alpha} + \frac{t_{0_\beta} - t_{0_\alpha}}{d_\beta} + \frac{d_\beta}{d_\alpha}. \tag{IV.3}$$

Using the following notations

$$D = \frac{d_\beta}{d_\alpha} \quad and \quad T = \frac{t_{0_\beta} - t_{0_\alpha}}{d_\alpha}, \tag{IV.4}$$

and since α_i is an integer, we obtain the following relation for the frames α_i of the master stream that should *play* while frame β_j of the slave stream plays

$$\alpha_i \in \begin{cases} [\beta_j D + T,\ \beta_j D + T + D - 1] & \text{if D, T} \in \mathcal{Z} \\ [\lceil \beta_j D + T - 1 \rceil,\ \lfloor \beta_j D + T + D \rfloor] & \text{otherwise} \end{cases} \tag{IV.5}$$

Relation (IV.5) gives the sequence numbers of the master stream frames that should play while frame β_j of the slave stream *plays*. In order to find the sequence number of the master stream frame during which β_j *starts* playing (as this was our

objective), among the frames computed with relation (IV.5), we take the one with the smallest sequence number.

$$\alpha_i = \begin{cases} \mathcal{J}_j D + T & \text{if D, T} \in \mathcal{Z} \\ \lceil \mathcal{J}_j D + T - 1 \rceil & \text{otherwise} \end{cases} \tag{IV.6}$$

To guarantee the maximum acceptable skew between the two streams, we compute the tolerance (see Table III.3) as

$$t_{\alpha\beta} = \begin{cases} \frac{A_{\alpha\beta}}{d_\alpha} - 1 & \text{if } \frac{A_{\alpha\beta}}{d_\alpha} \in \mathcal{Z} \\ \left\lfloor \frac{A_{\alpha\beta}}{d_\alpha} \right\rfloor & \text{otherwise} \end{cases} \tag{IV.7}$$

Therefore, frame $\mathcal{J}_j$ can *start* playing if the sequence number of the master stream frame currently playing, α_{play}, satisfies the condition

$$\alpha_i - t_{\alpha\beta} \leq \alpha_{play} \leq \alpha_i + t_{\alpha\beta} \tag{IV.8}$$

where α_i is computed using relation (IV.6).

As an example, assume that $d_\alpha = 50$ ms, $d_\beta = 66$ ms, the slave stream started 132 ms after the master stream and the maximum asynchrony between the two streams is 100ms. In this case, $D = 0.76, T = -132/66 = -2$. The currently playing master frame is 7. We want to know if frame 5 of the slave stream can start. Using relations (IV.6) and (IV.7) we find that the master frame 7 should play and the tolerance is 1. Since 7 is also the currently playing master frame, condition (IV.8) is satisfied, so slave frame 5 can play.

IV.2 The Lip-Synchronization

The network and hosts load variations may cause serious asynchrony between audio and video streams. In this section we propose and implement generic synchronization algorithms that take into account network and host load variations. We compare their

performances with the classical "drop-delay" algorithms [11, 12, 16, 20, 49], widely used in multimedia applications.

In the past, this problem has been studied in the context of *record and playback* of videoconferences which use *medium-size* windows (320 × 240 pixels) [11, 16, 43, 49]. In contrast, we consider *real-time* videoconferences that display video images in *large-size* windows (640 × 480 pixels). Though the challenges posed by real-time and record/playback applications in achieving synchronization are similar, there are several subtle differences.

1. Even in the absence of network and host load, for a 640 × 480 window, we have routinely measured a skew of 256 ms, which is significantly larger than the maximum acceptable value of +/− 160 ms recommended by Steinmetz [54]*.

2. The time to display a video frame in a large window can be significant. For example, from our experience, for 24 bits depth windows, all of the algorithms described in literature (see Chapter II) worked for 320 × 240 pixels windows, but did not work properly for 640 × 480 windows. This is because they do not estimate the display time of a video frame which in this situation is around 243 ms, again much larger than the acceptable skew.

With these observations we give our lip-sync algorithm (for notations see Table III.3):

$t_{estimated}$ = *InitialValue*; /* initialize the display time estimation */
while(1){
  getFrame (*v*): /* get video frame v from the application buffer*/
  t_d = *decompressFrame*(*v*): /* decompress the frame
    and measure the time */

*Our experimental setup was described in Section I.

a_{play} = *getCurrentlyPlayingAudio*() + $\left\lfloor \frac{t_{estimated}}{d_a} \right\rfloor$: /* compute *the audio frame that will play at the end of the display time of the video frame* */

a_t = *computeAudioShouldPlay*(v): /* compute the audio frame that should play if this video frame would start */

if($a_t < a_{play} - t_{av}$): /* video frame is behind */

case *VideoTrash* : /* a late frame is dropped */

continue;

case *VideoTrashAudioDelay* : /* a late frame is dropped:

if(*TrendVideoBehind*) /*delay audio if this is a trend */

delayAudio( ):

continue;

case *VideoNoTrashAudioDelay* : /* no frame is dropped:*/

if(*TrendVideoBehind*) /*delay audio if a trend */

delayAudio( ):

if($a_t > a_{play} + t_{av}$) /*video frame ahead,sleep */

sleep($(a_t - a_{play}) \times d_a$):

t_p = *play*(v)/* video frame on time, play ,measure display time */

$t_{estimated} = 0.25 \times t_p + 0.75 \times t_{estimated}$; /*compute a new estimation for the display time */

$v = \left\lfloor \frac{(t_d + t_p)}{d_v} \right\rfloor$: /*compute the sequence number of the next playing video frame */

}

We initialize the estimation of the video display time with a value off-line measured for that workstation. If such a value is not available, then we give it an arbitrary value (for example, 10 ms). To schedule a video frame for display, we do the

followings. First, we compute the sequence number of the audio frame that plays at the end of the video display time (a_{play}). We assume that audio plays continuously, so a_{play} has a sequence number which is $\lfloor \frac{t_{estimated}}{d_a} \rfloor$ greater than the currently playing audio frame. This assumption proved to be valid in our experiments as the audio process requires small computation times and it is scheduled more often that the video process. In the above formula, $t_{estimated}$ is the estimation of the display time computed after the previous frame has been displayed, (using the regression function III.6), and d_a is the duration of an audio frame. After that, we compute the sequence number of the audio frame that should play (a_i) if this video frame would start (using relation IV.6).

If a_i and a_{play} match (within the tolerated asynchrony), then the video frame is displayed and the display time is updated. If a_i and a_{play} do not match, then the video stream is either ahead or behind. If the video stream is ahead, then it sleeps for the time by which it is ahead and then it is displayed. If the video frame is behind, then the action we take depends on the protocol type.

We have evaluated four protocols, described in table IV.1. We compare these protocols by the way they handle the synchronization condition. Protocol P1 does not do anything when video and audio are out of sync. Protocol P2 (above, case *VideoTrash*) is the classical approach used in literature [11, 12, 16, 20, 49] for lip-synchronization: delay a video frame that is ahead and drop a video frame that is late. Protocol P3 (above, case *VideoTrashAudioDelay*) is our first protocol and it derives from P2, with the addition that it delays the audio stream if video tends to be behind. We estimate the asynchrony between audio and video by exponentially averaging with a smoothing factor of 0.9. When the estimated average asynchrony exceeds 160 ms, we delay audio. Protocol P4 (case *VideoNoTrashAudioDelay*) is our second protocol. Unlike P3 in this protocol we do not drop a video frame when it is late. However, similarly to P3 we delay audio if video tends to be behind.

Table IV.1: Specification of lip-synchronization protocols.

Protocol	Video Behind Audio Correction	Video Ahead Audio Correction
P1	do nothing	do nothing
P2	drop video	wait for matching audio
P3	(1) drop video (2) if this is a trend.delay audio	wait for matching audio
P4	if this is a trend.delay audio	wait for matching audio

From the user perceptive point of view, with P1, the skew is visible and the presentation is annoying. As we start skipping video frames, with protocol P2, the streams are synchronized, but the quality of the image is very bad, almost no motion. When we both skip video frames and delay audio (protocol P3), the quality of the image is better, but sometimes the image freezes for 3-4 seconds. With P4, where no video frames are dropped, the streams are synchronized and the quality of the image is very good.

IV.2.1 Implementation Issues

The receiver audio and video processes are implemented using two threads per process, with one thread as the producer (which receives and buffers frames arriving from the network) and the other one as the consumer (which plays the frames). This avoids internal UDP buffer overflow which may happen if a frame arrives early and the process sleeps. The two processes communicate with each other through a shared memory where the video process periodically writes the average asynchrony between audio and video. The audio process uses this information to know how long to delay an audio frame. We want to mention that we delay only audio frames that are the

first after a silence period.

IV.3 Synchronization of the Shared Windows Stream

In this section we describe protocols for integrating real-time audio, video and X-window streams in computer-supported cooperative environments. The X requests (shared windows packets) generated by an X client are sent to the local and remote X servers. Ideally, all the X servers receive and play the X requests at the same time, while the audio and video devices receive and play the audio and video frames at the same time. In practice, due to the best-effort nature of the current operating systems and networks, and due to the heterogeneity in workstations performances, there can be large skews between audio, video and X windows.

IV.3.1 Key Considerations

While audio and video streams are stateless, periodical and continuous, the X windows stream is stateful, aperodical, and discrete. Due to these differences we cannot apply directly the synchronization protocols we have developed for audio and video. More precisely, due to the stateful nature of the X-windows stream dropping or/and duplicating an X-request is usually not permitted.

The main challenge in synchronizing the X windows stream is the large amount of time it takes the X server to process some X requests. For example, it takes almost 395 ms to create and configure a window, and around 475 ms to display a 3 square inches image in Netscape. A simple correction like dropping X requests in the case of asynchrony, is not always enough as not all X requests can be dropped (e.g., *FreePixmap* can be dropped, but *CreateWindow* cannot). For this reason, our protocols gradually increase the number and type of corrections applied to the streams in order to keep them synchronized. If the streams are not synchro-

nized, we first drop as many X requests as we can. If this is not enough to keep the streams synchronized, then we also delay the X client.

Our synchronization protocols use a master/slave model. When audio is present, X windows is synchronized after audio (the same is true for video), i.e., audio is the master stream. When there is no audio, we synchronize video after the X windows stream. In this situation, X windows is the master stream. We choose to synchronize the X windows stream after the audio stream because audio has stringent jitter and latency requirements and delaying audio more than necessary will result in noticeable discontinuities. The X windows stream, on the other hand, typically does not have such temporal requirements and can be delayed for synchronization purposes. When no audio stream is present, we synchronize video after the X windows stream, as it is easier for video to catch up after X windows than it is for X windows to catch up after video. This is because video frames can be dropped. The synchronization algorithm between video and the X windows is similar with the lip-synchronization one.

According to Steinmetz [54], the user acceptable skew between audio and video is (-160, +160) ms, while the user acceptable skew range for audio and X windows is (-500, +750) ms. On the other hand, there is no accurate measurement of the user acceptable skew between X windows and video, mainly because the two streams are uncorellated, unless the video image captures the image on the X server. In this situation, from our experience, unless it is a very specialized video camera that allows to set its vertical scan rate to match the monitor, you will see the monitor at the destination leading scan lines as the image is drawn on the screen, so it would be impossible to follow up the synchronization between the streams. With these considerations, we assume that video and the X windows are uncorrelated. When all three streams, i.e., audio, video and X windows are present, we take audio to be the master for the other two streams. When audio is not present, we synchronize video

after X windows.

To establish the tolerance range of the asynchrony between video and X-windows, we use the relations experimentally determined by Steinmetz [54]:

$$-160ms \leq async_{av} \leq +160ms \qquad (IV.9)$$

$$-500ms \leq async_{ax} \leq +750ms \qquad (IV.10)$$

where $async_{av}$ is the asynchrony between audio and video and $async_{ax}$ is the asynchrony between audio and X-windows. Consequently, the asynchrony between video and X-windows has to be within the range [-660, +910] ms.

IV.3.2 The Synchronization Algorithm

Our first goal is to identify the X requests that can be dropped. Clearly, it is not possible to drop any request that creates a resource, since future requests may try to refer that resource. On the other hand, it seems reasonable to be able to drop requests that just draw on the screen. To identify what other types of requests can be dropped we have tested the effect caused by dropping them on the following typical applications: xterm, emacs and Netscape. Based on the application behavior, we have identified the following categories of X requests (see *Appendix A*):

1. **Requests that crash the X client if dropped.** This category consists of (1) requests that create resources (windows, pixmaps, cursor, e.g. *CreateWindow*, *CreatePixmap*, *CreateCursor*), (2) modify the properties of existing resources (e.g. *ChangeWindowProperties*, *ChangeGC*), (3) change window position in the X-server hierarchy (e.g. *ReparentWindow*), and (4) requests that grab the pointer and the keyboard (*GrabButton*, *GrabKey*).

2. **Requests that freeze the X client if dropped.** These are the requests that query the X server and wait for an answer back. The X client is not doing further processing until the answer gets back. Thus if the query is not sent to the X server, no answer is received and the X client blocks. Examples of such requests are *GetWindowsAttributes*, *QueryTree*, *TranslateCoordinates*.

3. **Requests that affect other X clients if dropped.** For example, if *UngrabPointer* request is dropped, the user cannot move the mouse in a window different than the one which grabbed the pointer. If *ChangeHosts* request is dropped, and the request adds a host to the access list, then that host cannot connect to the X server.

4. **Requests that can be safely dropped.** In this category enter the requests that (1) destroy resources (e.g. *DestroyWindow*, *FreeGC*, *FreePixmap*), (2) manipulate windows by the X client (e.g. *MapWindow*, *UnmapWindow*), (3) draw graphics (e.g. *PolySegment*, *PolyRectangle*, *PolyFillRectangle*), (4) print text (e.g. *PolyText8*, *PolyText16*, and (5) put images (*PutImage*).

Out of the 120 requests documented by the X Consortium [46], 21 are requests that crash the X client if dropped, 43 are requests that freeze the X client if dropped, nine are requests that affect other X clients if dropped and 47 are requests that can be safely dropped. Our policy is to drop only the requests that do not affect in any way other applications. Consequently, we drop only the requests in the last category, i.e., a total of 47 requests.

With these considerations, the synchronization algorithm between the X windows, audio and video streams is the following.

```
while(1){
    getPacket (x): /* get X packet from the application buffer; */
```

Table IV.2: Specification of X-windows synchronization protocols.

Protocol	Windows Behind Audio Correction	Windows Ahead Audio Correction
X1	do nothing	do nothing
X2	drop X request, if possible	wait for matching audio
X3	if this is a trend, delay the X client	wait for matching audio
X4	(1)drop X request, if possible (2) and delay the X client	wait for matching audio

```
case Xmaster : /* audio not present, X windows is master*/
    playXpacket(): /* video is synchronized after X windows*/
case Xslave : /* X windows and video are synchronized after audio */
    a_play = getCurrentlyPlayingAudio():
    a_i = computeAudioShouldPlay(x):
    if(a_i < a_play - t_ax):/*X windows is behind*/
        case SkipXreq : /* drop the request, if possible */
            if(requestAmongDrop):
                continue:
        case DelayXClient :/* if during the last MIN_REQUESTS packets
                              audio and video are out of sync, send a delay message
                              to X client to stop sending packets */
            if(packets > MIN_REQUESTS)
                send DELAY_Xclient message to sender
                    packets = 0:
            else
                packets++;
```

```
case SkipXreqDelayXClient : /* drop a request, if possible:
                if the asynchrony is larger than ASYNC_MAX
                for more than MIN_REQUESTS packets
                delay the X client*/
        if(requestAmongDrop):
          drop it:
        if(async > ASYNC_MAX)
            if(packets > MIN_REQUESTS)
                send DELAY_Xclient message to X sender:
                packets = 0:
            else
                packets++:
    if(a_i > a_play + t_av) /*X windows ahead, sleep */
        sleep((a_i - a_play) x d_a):
}
```

When an X request arrives, first we retrieve the sequence number of the current playing audio frame, a_{play} , from the audio device. Then we use relation (IV.6) to compute the sequence number, a_i, of the audio frame that should be played when the X request starts. If the X request is ahead, then the process sleeps for a duration of time equal to the current asynchrony between audio and X windows. If the X windows is behind, the action depends on the protocol type.

We have investigated four synchronization protocols for audio and X windows (see Table IV.2). Protocol X1 does not perform any synchronization, and therefore we use it as a baseline comparison. In X2 (above, case *SkipXreq*), if an X request is late and it can be safely dropped, then it is ignored. In all the other cases the request is served†. On the other hand, if an X request is ahead, it is delayed until

†Since the image may get very fuzzy due to many packets being dropped, we force an X expose

the corresponding audio arrives.

In X3 (above, case *Delay.XClient*), no X request is dropped. However, if the host processor cannot keep pace with processing the X stream, i.e., the X stream is consistently behind audio, then the sender is asked to slow down. To accomplish this, if the asynchrony is *persistent* the receiver sends a special message $DELAY_{Xclient}$ containing the current asynchrony to the sender. In turn, the sender uses this value to compute an estimated asynchrony (by exponential averaging). If the estimated value is larger than the acceptable asynchrony the X client sleeps. We say that the asynchrony is persistent if for more than $MIN_{REQUESTS}$, the asynchrony is greater than a certain threshold $ASYNC_{max}$. Finally, protocol X4 (above, case *Skip.XreqDelay.Xclient*) combines both techniques used in protocols X2 and X3. We note that we also tried to delay audio when X-windows lags behind, (similar with our approach for video, see Section IV.2, but the results were not encouraging. The main reason is that the skews between X stream and audio are much larger than between video and audio, and delaying audio for such a long interval makes the presentation annoying.

Unlike the lip-synchronization algorithm presented in Section IV.2, for X-windows we ignore the display time of an X windows request. Although we have determined that the display time of some X requests is fairly large, e.g., *PutImage* may take 475 ms, the main reason for this strategy is the fact that the display time of the same X request varies so much according to the parameters of the request (see Section III.3 which makes it impossible to predict. Instead, our approach is to measure the asynchrony between X windows and the other streams after the request has been served and apply corrections, such that within a short time interval streams will be in sync again.

---

event periodically when there is no X activity. Also if the dropped request is within the categories *DestroyResources, Keyboard and Pointer* except *WarpPointer*, or *Miscellaneous* (see Appendix A), at this time we send it to the X server.

IV.4 Summary

The multimedia synchronization task always arises when a variety of media with different temporal characteristics are brought together and integrated into a multimedia system. In order to synchronize the streams, the first requirement is to establish a synchronization condition. Existing synchronization conditions restrict media units to have the same duration, or durations that have a common divisor, which limits its flexibility. Approaches that use temporal timestamps waste network bandwidth. Moreover media units already have sequence numbers as they use the services of unreliable transport protocols. We propose a simple synchronization condition based on sequence numbers which allows streams to have different media unit durations.

Based on our synchronization condition, we introduce our algorithm for lip-synchronization. Our algorithm works for large size video windows, a case when existing solutions fail due to the fact that they do not estimate the display time of a video frame.

In this chapter we have also introduced our algorithms for synchronizing the shared windows stream. Existing solutions, either delay audio stream when the shared windows stream is behind, or modify the rate of sending shared windows packets to the X server. From our experience, in the case of heavy user interaction with the X application, there is a very large skew between audio and X windows and delaying audio with such a long interval, introduces sensible discontinuities in the audio stream. Modifying the rate of sending shared windows packets to the X server may also not achieve the synchronization purpose, as the rate of processing X requests depends on the processing rate of the X server. In the case of high user interaction, this lags far behind audio. Our synchronization algorithm combines both skipping X requests with delaying the X client and achieves synchronization even in the cases of high user interaction.

Chapter V

Media Synchronization in Distributed Systems

"All truths are easy to understand once they are discovered; the point is to discover them."

Galileo Galilei

It is often the case that multimedia applications involve more than two users at the same time. For example, in a distance learning application, usually there is a teacher and a number of students. Media synchronization in this situation poses two additional problems: (1) to extract the synchronization information from mixed audio streams and (2) to provide a global order of events.

The first problem appears when audio streams originating from different users arrive at the destination at the same time (e.g., if two students speak at the same time). Since every workstation has only one audio device, the audio streams need to be mixed before they are played. Unfortunately, by mixing the audio streams, the synchronization information is lost (see Chapter I for an example). Therefore, at the destination, we need not only to mix the audio streams, but also to extract the synchronization information from mixed audio streams.

The second problem arises when the multimedia application needs to be recorded and played back. In this situation, a mechanism that provides a global order of events in the system is necessary. At record time, this mechanism ensures

that the order of events occurring on different workstations is correctly stored in the record file. At playback time, events are played using the information in the record file.

Existing solutions to the temporal synchronization problem consider that only one audio stream arrives at the destination, at one time. In this chapter, in Section V.1 we present the mechanism of extracting the synchronization information from mixed audio streams. This mechanism provides the required extension of our algorithms to work in a distributed system. Next, in Section V.2 we describe a simple algorithm that achieves a common time for a distributed multimedia application. Existing solutions create a single point of failure, and thus they are more limited (see Chapter II).

V.1 Extracting the Synchronization Information from Mixed Audio Streams

The problem that we want to address derives from the fact that incoming audio streams need to be mixed before being played (see Chapter I for a detailed description of the problem and an example). Unfortunately, by doing so we lose the synchronization information between indexes of a particular audio and video streams. More precisely, for each video frame we need to know the index of the corresponding audio packet that is playing. However, after mixing, the sequence number of the playing audio packet may no longer match the sequence number of the audio frame of the stream we are interested. Furthermore, the audio packet that is currently playing may not contain any data from that stream. Bellow we describe our solution to this problem.

Every audio and video packet is described by the following format: (*userId*, *streamId*, *seqNumber*, *data*), where *userId* represents the user identifier which is

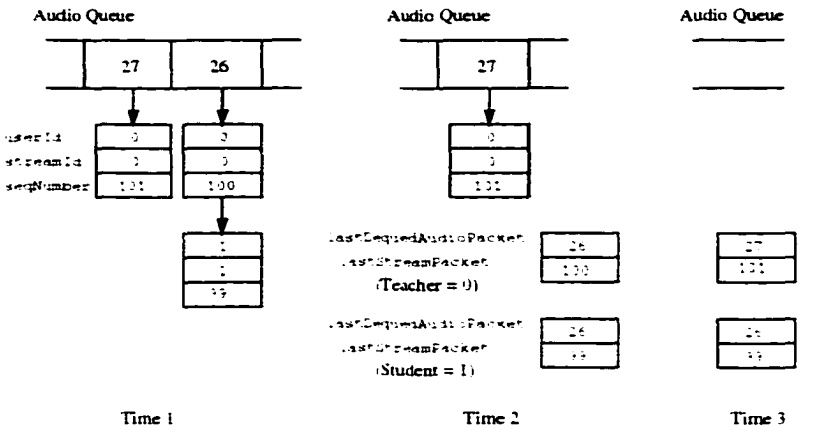


Figure V.1: The packet queue and the values of *lastDequedAudioPacket* and *lastStreamPacket* variables for two audio streams at three time instances.

unique with respect to an application. *streamId* represents an identifier assigned to each stream originated from that particular user, and *seqNumber* represents the sequence number assigned at the sender by our algorithm. *streamId* is unique with respect to all audio streams originated from the same sender*.

At the receiver, the mixed audio packets are stored in a special purpose queue. With each entry in the queue we associate a list containing information about the audio streams whose packets were mixed in that entry. More precisely, each element in the list contains the same information as the corresponding packet, excepting audio data, i.e., (*userId*, *streamId*, *seqNumber*). In addition, associated with each audio stream we maintain two variables: *lastDequedAudioPacket* and *lastStreamPacket*, where *lastDequedAudioPacket* indicates the sequence number of the last packet from the queue that has been sent to the device and contains the packet with the sequence number *lastStreamPacket* of that audio stream.

*Here we assume underlying transport protocols that do not carry the *userId*, *streamId*, *seqNumber* information (e.g., UDP). If the transport protocol provides this information (e.g., RTP), we no longer need to store it in audio/video headers.

For clarity, consider the following example: assume a session with one teacher and one student. Both the teacher's *userId* and his audio *streamId* are 0. Similarly, the *userId* of the student and his audio *streamId* are 1. Figure V.1 shows the state of the queue and the value of the *lastDequeudAudioPacket* and *lastStreamPacket* variables at three consecutive instances of time. Initially, assume that the queue contains two packets: the first packet consisting of frame 100 of teacher's audio, and the audio packet 99 of the student, and the second packet consisting of the teacher's 101 packet only. Also, assume that so far the audio process has sent 25 packets to the audio device. Thus, the corresponding indices of the two packets in the queue are 26, and 27. At the next instance of time assume that the audio process sends the next packet (with index 26) to be played. Consequently, the *lastDequedAudioPacket* variables of both the teacher's and student's audio streams are set both to 26, while their *lastStreamPacket* variables are set to 100 and 99 respectively. Next, at the second instance of time, when the packet 27 is sent to the audio device, only *lastDequedAudioPacket* of the teacher's audio stream is changed to 27 and its *lastStreamPacket* is set to 101; the corresponding variables associated to the student's audio stream remain unchanged since none of its packets is mixed in packet 27.

Further, given an audio stream it is straightforward to determine the sequence number of its current playing frame seq_{play}. More precisely, we have:

$$seq_{play} = lastStreamPacket - (lastDequeuedAudioPacket - getCurrentlyPlayingAudio()), \quad \text{(V.1)}$$

where obviously *lastStreamPacket* and *lastDequeuedAudioPacket* are the variables associated to the current audio stream. Consider again the example in Figure V.1.

Assume that at the third instance of time we want to get the current playing audio packet of the student's stream. Assume that $getCurrentlyPlayingAudio()$ returns 24.[†] Then, by using the above equation we have: $seq_{play} = 99 - (26 - 24) = 97$. It is worth noting that a more accurate solution would be to remove the entries from the queue only *after* that packet has been played by the audio device. However, this will complicate the algorithm and will increase the buffer requirements, while, as we have observed in our experiments, improving little the accuracy. Another variation of the algorithm would be simply to set the $lastDequeuedAudioPacket$ and $lastStreamPacket$ as soon as the packet is mixed. Although this results in a much simpler data structure (we no longer need lists associated to each packet), the potential inaccuracy generated by the eventual audio device buffer overflow can be quite large. Therefore, the solution we chose can be viewed as a tradeoff between the complexity and accuracy.

With these considerations, to extend our lip-synchronization solution to a distributed environment, we need to do the followings:

1. In the shared memory between audio and video, we also store:
   - lastPlayedAudioPacket, the sequence number of the last audio frame sent to the audio device.
   - lastStreamPacket[*id*], the sequence number of the last frame sent to the audio device for the audio stream coming from user *id*.
2. The sequence number of the audio frame that will play when the video frame starts is given by

$$a_{play} = seq_{play} + \left\lfloor \frac{t_{estimated}}{d_a} \right\rfloor . \qquad (V.2)$$

[†]This means that the queue of the audio device stores at this point the packets 25, 26, and 27.

where seq_{play} is computed with relation (V.1), and $t_{estimated}$ is the estimated display time of the video frame, computed with relation (III.4).

V.2 A Common Time System for a Multimedia Application

To achieve a common time system, our approach uses the concept of "time frame" introduced by Li and Ofek [25]. The time is divided into discrete time units referred to as time frames. Each workstation has a local counter which is incremented at the start of each new frame. Ideally, we would like that all workstations to start the first time frame (local frame counter 0) at the same time. A straightforward solution would be to use a global clock mechanism, such as NTP [36]. Unfortunately, this imposes a high overhead. In addition, since the accuracy of our synchronization algorithm is of the order of a frame duration, an algorithm that synchronizes the staring times with an accuracy of 10-20 ms would be acceptable. In the remaining of this section we propose a simple distributed algorithm to achieve this goal. In short, when a new workstation joins the group, it asks the other members in the group, if any, about their starting times. Upon receiving a certain number of "good" replies, it averages over the resulting values and compute its starting time (a "good" reply is a reply with a low round-trip time). Note that our algorithm is totally decentralized in the sense that it does not assume a master workstation that keeps the reference time. This is in order to increase both the robustness and the generality of our solution. The averaging mechanism to compute the starting time is intended to avoid the error propagation as more and more workstations becomes active.[‡] Following we give the algorithm details.

[‡]Consider the case of n workstations that become active sequentially, and assume that each of them gets the starting time from the previous workstation that has became active. In this way the error between the first workstation that became active and the last one is proportional to the number of active workstations, which for a large n would be unacceptable.

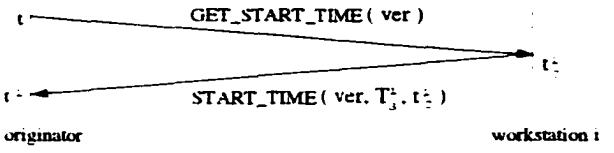


Figure V.2: The time diagram for evaluating the starting time.

When a workstation joins the conference, it multicasts a GET_START_TIME message. (In the remaining of this section this workstation is also called *originator*.) Let t be the time when this message is sent. Upon receiving such a message, every machine i replies with a START_TIME message containing the start time T_s^i of that machine, and the time t_c^i when the reply was sent. If the starting time of a machine is not set yet, then the GET_START_TIME message is simply ignored. Upon receiving a reply, the originator first compute the time t^i when the message has been received. Then, it uses the following formula to compute its local starting time $T_s'^i$ based on the information received from the i-th machine:

$$T_s'^i = T_s^i + \frac{t - t^i}{2} + t - t_c^i. \tag{V.3}$$

Figure V.2 shows the time diagram used in deriving the above equation. Similarly to V.2 we assume that the latency for both GET_START_TIME and START_TIME messages is the same. More precisely, let Δ_i denote the time slack between the originator and machine i. That is, when the time at the originator is t, the time at workstation i is $t + \Delta_i$. Since the message latency is assumed to be symmetric we have:

$$t + \frac{t - t_i}{2} = t_c^i + \Delta_i. \tag{V.4}$$

From the above equation and the fact that $T_s'^i = T_s^i + \Delta_i$, Eq. (V.3) results immediately. In order to minimize the effect of network latency and CPU load variations,

the originator computes its starting time by averaging over multiple T_s' values. In addition, to eliminate the effect of packet losses, only the replies for which the round trip time (i.e., $t_i - t$) do not exceed a certain threshold are considered. To achieve a reasonable accuracy the originator waits for N "good" replies before computing the average. If after sending the first GET_START_TIME message, the originator does not receive N replies, it keeps resending it until it eventually receives N replies. To differentiate between a new reply and a late reply to a previous GET_START_TIME, each message has a version number that is incremented every time the originator sends a GET_START_TIME message. To break the eventual "synchronization" between two workstations that may join an "empty" group simultaneously, the time-out value is uniformly distributed between TO_START_TIME and 2×TO_START_TIME. In our experiments we use $N = 10$, TO_START_TIME = 30, and a 20 ms threshold for $t - t_i$, which proved to be large enough for our extended LAN setting. This guarantees that the eventual errors in determining the starting time will be several times smaller than the duration of a video or an audio frame.

variables:

$ver = 1$, $T_s = 0$, $ctime = 0$, $msg\_cnt = 0$, t;

on joining conference:

GET_START_TIME.$ver = ver$;

$t = getCrtTime()$;

multicast(GET_START_TIME);

setTimeOut(TO_START_TIME);

on receiving message :

case GET_START_TIME:

if ($T_s > 0$) {

/* the local machine has computed T_s; send a reply */

```
            START_TIME.ver =GET_START_TIME.ver;
            START_TIME.t_c = getCrtTime();
            START_TIME.T_s = T_s;
            sendReply(START_TIME);
        } else if (msg_cnt = 0)
            restart algorithm;
    case START_TIME:
        if (T_s > 0 or ver ≠START_TIME.ver)
            /* if starting time already computed or this is a late reply, ignore it */
            break;
        t_1 = getCrtTime();
        if (t_1 − t < MAX_RTT) {
            /* compute starting time using Eq. (V.3) */
            T'_s =START_TIME.T_s + (t + t_1)/2−START_TIME.t_c;
            ctime = ctime + T'_s;
            msg_cnt++;
            if (msg_cnt == N)
                T_s = ctime/msg_cnt; /* compute starting time */
                return;
        }
on TO_START_TIME time-out:
    ver = ver + 1;
    if (ver >MAX_VER)
        if (msg_cnt == 0) /* this is the first workstation joining the group */
            T_s = getCrtTime();
        else {
```

```
            T_s = ctime/msg_cnt; /* compute starting time */
            return;
        }
    } else {
        /* re-send GET_START_TIME message */
        GET_START_TIME.ver = ver;
        t = getCrtTime();
        multicast(GET_START_TIME);
    }
}
```

To determine the constant values in the above algorithm we have conducted several experiments over an extended LAN consisting of 20 computers located in two sites (Norfolk and Virginia Beach) 20 miles one of each other. We measured the round-trip time at the application level among the workstations at the same site, as well as between workstations at different sites (for a description of our testbed, see Chapter 1). To get realistic data, all the experiments were conducted during class time with all workstations running the IRI software. Between two workstations situated at the same location we have measured an average round-trip of 3.55 ms with the coefficient of variation 1.08. Similarly, the average of the round-trip time between two workstations situated at different locations was 9.93 ms with the coefficient of variation 0.78. For obtaining these data we have conducted over 1500 individual measurements. Based on these results we have chosen the threshold MAX_RTT to be 20 ms, and TO_START_TIME to be 40 ms.

Let T_s be the start time computed by originator. Then each workstation will keep a virtual clock that starts at time T_s, and which is incremented every Δ real time units. The common time can be viewed as a stream with frame duration Δ, that

plays on all workstations and that has started simultaneously on all workstations. Therefore, we call it *clock stream*. After the clock stream has started, all the events in the system are related to it, as follows.

Every X request is timestamped with the sequence number of the clock stream. To relate the sequence numbers of the continuous streams to the *clock stream* sequence numbers, we divide a sequence number computed with the above algorithm, to the clock stream frame duration.

$$\alpha_c = \left\lfloor \frac{\alpha_c \times d_{alpha}}{\Delta} \right\rfloor. \qquad (V.5)$$

We note that by relating the sequence numbers to the clock stream sequence numbers, all the streams have the same frame duration, which equals Δ.

V.3 Summary

In this chapter, the objective was to extend our synchronization algorithms to work in a distributed system. We achieve this by providing (1) a mechanism that extracts the synchronization information from mixed audio streams and (2) a protocol that creates a lightweight common time in a distributed system.

To extract the synchronization information, for each stream we keep two variables, *lastDequedAudioPacket*, and *lastStreamPacket*, where *lastDequedAudioPacket* indicates the sequence number of the last packet from the audio queue that has been sent to the device and contains the packet with the sequence number *lastStreamPacket* of that audio stream. Then, for each audio stream, the sequence number of its frame mixed in a particular audio packet can be obtained by subtracting from the *lastStreamPacket* variable, the *lastDequeuedAudioPacket* variable and the current playing audio packet obtained by polling the audio device.

To achieve a common time in a distributed system, our approach is as follows.

When a new workstation joins the group, it asks the other members in the group, if any, about their starting times. Upon receiving a certain number of "good" replies, it averages over the resulting values and compute its starting time (a "good" reply is a reply with a low round-trip time). Our algorithm is totally decentralized in the sense that it does not assume a master workstation that keeps the reference time. This is in order to increase both the robustness and the generality of our solution.

Chapter VI

Effect of Network Load

"There are three principal means of acquiring knowledge...
observation of nature, reflection and experimentation.
Observation collects facts; reflection combines them;
experimentation verifies the result of that combination."

Denis Diderot

An important factor that influences users' perception of a multimedia application is the network load variation. High load on the network determines an increase in the end-to-end latency of communication between participants, an increase in the number of discontinuities (i.e., frames are either never played or played multiple times) and a deviation from the exact synchronization between audio, video and X-windows.

In this chapter we present the experiments we performed in order to validate our synchronization protocols in the presence of network load. The network configuration used for experiments is showed in Figure VI.1. It consists of an extended LAN with 20 Sun computers located in two sites, Norfolk and Virginia Beach, 20 miles away one from each other (see Chapter 1 for a detailed description of the testbed). To put load on the network, on a workstation we run a program that periodically (every 40 ms) sent packets to another workstation. The program takes as argument

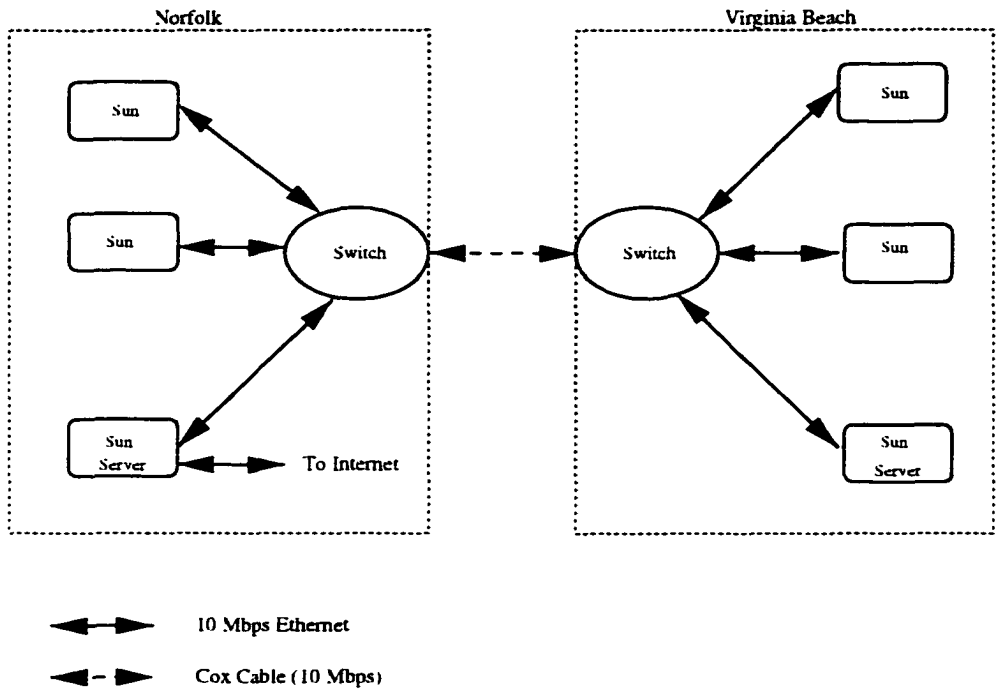


Figure VI.1: The Network configuration.

the load to be put on the network. Based on this, it computes the packet size *.

In Section VI.1 we present experimental results and evaluation of our lip-synchronization algorithms. After that, in Section VI.2 we describe the experiments that evaluate the synchronization of the shared windows stream with audio.

VI.1 Lip-Synchronization

For the lip-synchronization we have evaluated four protocols, described in detail in the previous chapter. Just to remind here, protocol P1 does not do anything when video and audio are out of sync. Protocol P2 delays a video frame that is ahead and drops a video frame that is late. Protocol P3 is similar with P2, with the addition that it delays the audio stream if video tends to be behind. In protocol P4, we do

*For example, for a 1 Mbps load, the packet size is 5,000 bytes.

Table VI.1: Percentage of audio and video frames successfully delivered at the destination in the presence of heavy network load.

Network load [Mbps]	Audio frames	Video frames
8	95%	95 %
8.25	94 %	94 %
8.5	87 %	56 %
8.75	84 %	52 %

Table VI.2: Percentage of video frames skipped with protocols P2 and P3.

Network load [Mbps]	in the case of P2	in the case of P3
8	84.6%	17.03 %
8.25	85.3 %	12.9 %
8.5	84.9 %	15.4 %
8.75	86.1 %	18.2 %

not drop any video frame that is late and we delay audio if video tends to be behind.

VI.1.1 Experiment Description

For each protocol we put on the network loads varying from 1 Mbps to 8.75 Mbps. We stopped at 8.75 Mbps, as for higher loads, the quality of the image became extremely poor and we were getting many NFS errors. From 1 to 8 Mbps loads, we increased the load by 1 Mbps each time. As we were getting significant difference in performance for loads larger than 8 Mbps, we also performed experiments with 8.25 and 8.5 Mbps loads.

Table VI.3: Evaluation of the asynchrony between audio and video in the presence of heavy network loads [number of audio frames].

Protocol	Network load [Mbps]	Medium value	Standard deviation	Variance	Skew out of range
P1 (do nothing)	8	-4.97	0.88	0.78	95.70%
	8.25	-4.91	1.20	1.46	88.78%
	8.5	-3.61	3.81	14.56	68.16%
	8.75	-0.52	5.21	27.22	59.13%
P2 (skip/delay video)	8	-2.02	0.79	0.62	0%
	8.25	-2.01	0.86	0.75	0%
	8.5	-0.95	1.92	3.69	0%
	8.75	-0.09	2.34	5.51	0%
P3 (skip/delay video delay audio)	8	-0.63	0.83	0.70	0%
	8.25	-0.55	1.85	3.44	0.5%
	8.5	-0.38	2.14	4.60	2.36%
	8.75	-0.27	2.31	5.31	4.78%
P4 (delay video delay audio no video skip)	8	-1.80	0.63	0.407	0%
	8.25	-1.75	1.21	1.48	4.57%
	8.5	-0.91	2.44	5.96	9.25%
	8.75	-0.41	2.62	6.91	11.11%

VI.1.2 Results and Evaluation

For each protocol, the evaluation metrics are:

1. The skew between audio and video *after* the X function that displays the video frame (*XShmPutImage*) has completed, as this gives the correct skew between audio and video streams.

2. The number of video frames, as a percentage of the total video frames received at the destination, that have skews out of the accepted range.

3. The number of video frames skipped, as a percentage of the total number of video frames received at the destination.

In addition to these measurements, we also determined the number of audio and video frames received by the destination as a percentage of the total number of frames sent by the source. This quantifies how much the quality of the application degrades due to packet loses in the presence of high load.

As mentioned, we measure the skew (asynchrony) between audio and video *after* the video frame has been displayed (at the end of the video display time). We do this by subtracting from the corresponding sequence number of the audio frame that should have played (using relation IV.6), the sequence number of the currently playing audio frame (obtained from the audio device). A negative skew indicates that video is behind, while a positive skew indicates that video is ahead. Throughout this chapter, we present the skew measured only in number of audio frames. If desired, the skew measured in milliseconds can be computed by multiplying the previous value by the audio period (64 ms). Like Steinmetz [54] we consider the skew acceptable as longs as it falls within the range (-2.5, 2.5) or (-160, 160) ms.

Since we did not notice any difference in these parameters for loads smaller than 8 Mbps, we present here the results for 8, 8.25, 8.5 and 8.75 Mbps loads.

Figures VI.2 - VI.5 show the results for each protocol. Tables VI.1 and VI.2 show the percentages of audio and video frames arriving at the destination, and the percentage of video frames skipped with protocols P2 and P3. Table VI.3 shows the average skew, its standard deviation and variance and the number of times the skew falls out of the accepted range in the presence of various network loads.

At 8 Mbps, with P1 we measured an average skew of −4.97 (318 ms) caused by the fact that audio is ahead of video. From the user perceptive point of view, the skew is visible and the presentation is annoying. As we start skipping video frames (with protocol P2), video catches up and the skew decreases to an average of −2.02 (128 ms). The streams are synchronized, but the quality of the image is very bad, almost no motion. When we both skip video frames and delay audio (protocol P3), the average skew becomes -0.63 (40 ms). The quality of the image is better, but sometimes the image freezes for 3-4 seconds. With P4, where no video frames are dropped, the skew is around -1.80 (115 ms) and the quality of the image is very good.

As load is introduced in the network, the cases when audio is ahead of video and behind of video, alternate. As practically there is a dedicate link between the two machines we run experiments on (see Figure VI.1), we believe that this happens due to the fact that both audio and video are queued in the switch before they are sent to the destination. The standard deviation and the variance increase with the load, but the average asynchrony decreases as the number of instances with negative skews offsets the one with positive skews.

Surprisingly, in the case of P1, the number of instances in which the skew is out of range decreases with the load. More precisely, it decreases from 95.70%, when the load is 8 Mbps to 59.13 %, when the load is 8.75 Mbps. This behavior is probably a result of the extra time spend by the audio and video frames in the switch buffers.

As expected, the more load is put on the network, the more frames are dropped by the switch. For 8 and 8.25 Mbps loads, almost the same percentages of frames are

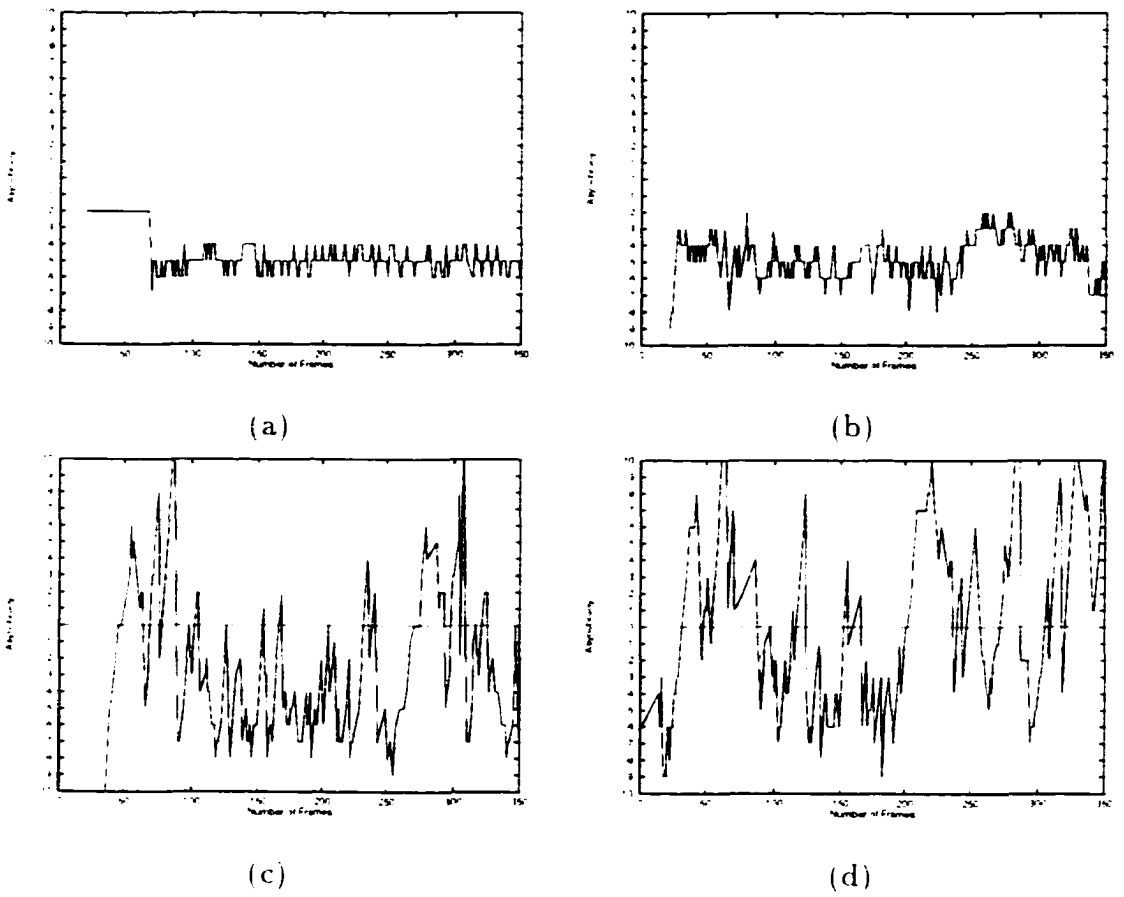


Figure VI.2: Variation of the skew between audio and video with protocol P1 (no correction), when a load of (a) 8 Mbps, (b) 8.25 Mbps, (c) 8.5 Mbps and (d) 8.75 Mbps was put on the network.

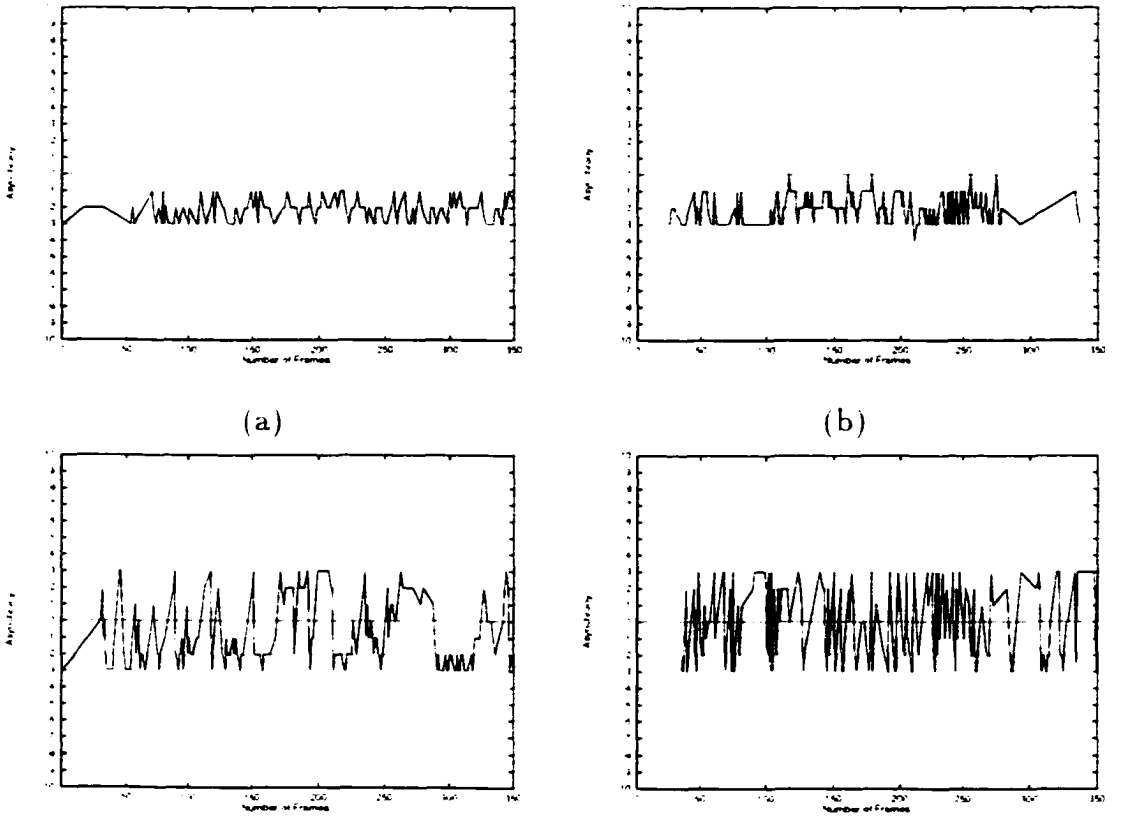


Figure VI.3: Variation of the skew between audio and video with protocol P2 (skip a late video frame, delay an early video frame), when a load of (a) 8 Mbps, (b) 8.25 Mbps, (c) 8.5 Mbps and (d) 8.75 Mbps was put on the network.

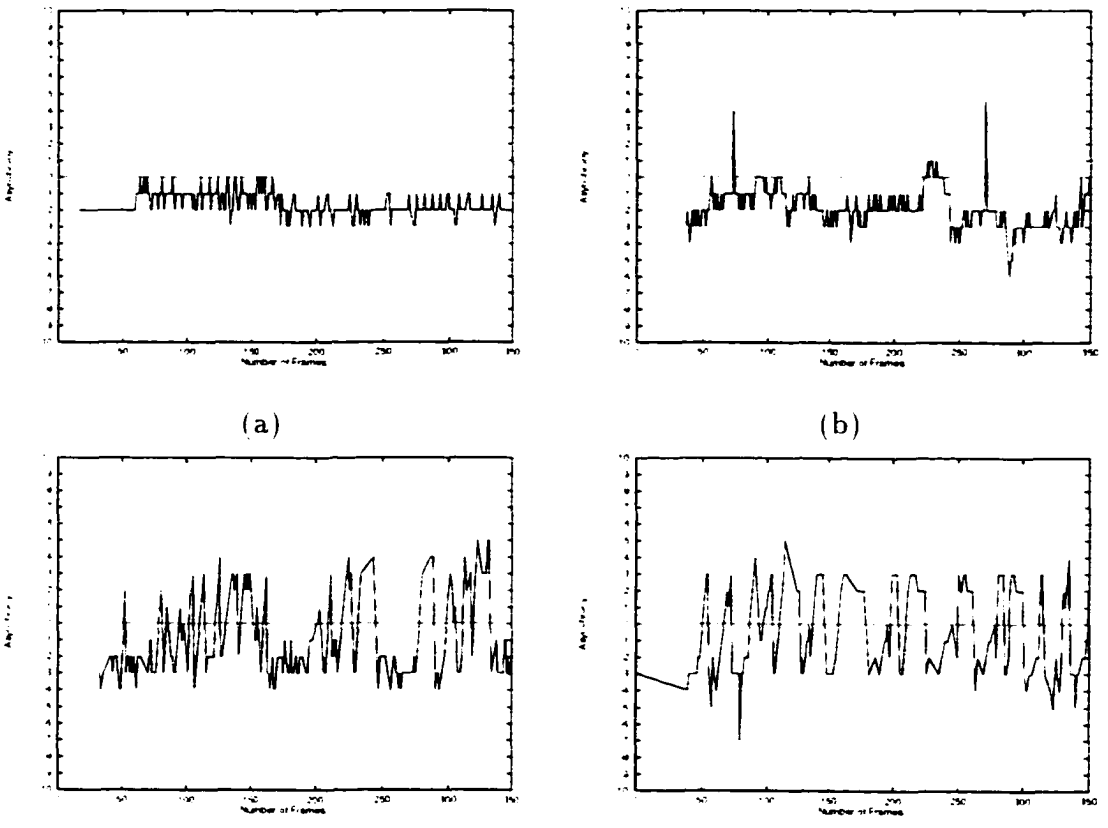


Figure VI.4: Variation of the skew between audio and video with protocol P3 (delay an early video frame, delay audio if it is a trend for video to be behind, no video skip), when a load of (a) 8 Mbps, (b) 8.25, Mbps, (c) 8.5 Mbps and (d) 8.75 Mbps was put on the network.

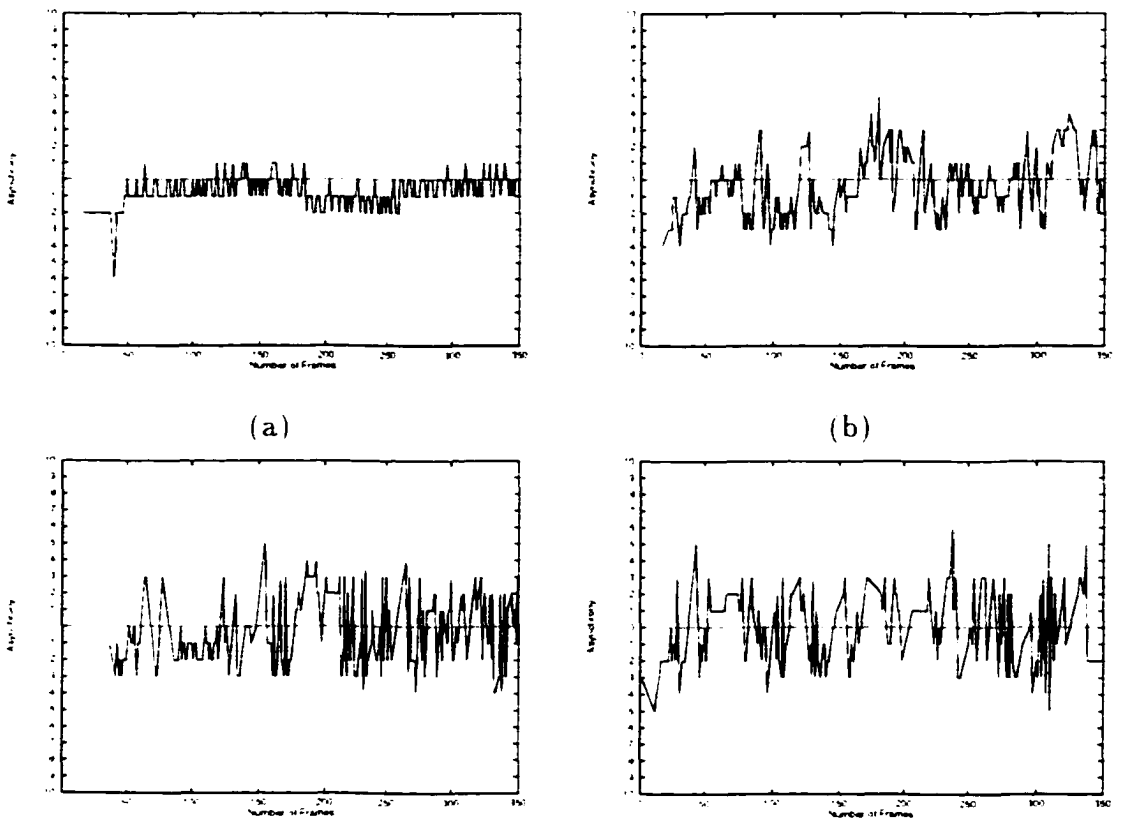


Figure VI.5: Variation of the skew between audio and video with protocol P4 (no video skip, delay video if it is behind, delay audio if it is a trend for video to be behind), when a load of (a) 8 Mbps, (b) 8.25 Mbps, (c) 8.5 Mbps and (d) 8.75 Mbps was put on the network.

received by the audio and video processes. However, for higher loads, the number of video frames decreases significantly due to the fact that the size of a video frame (varying from around 1KB up to 8.5 KB) is larger than the size of an audio frame (512 bytes). As the maximum Ethernet packet size is 1.5 KB, a video frames is usually divided in packets and sent over the network. Assuming that the probability to lose a packet is p it follows that an audio frame is lost with probability p (because it fits in one packet), while a video frame that is divided over n packets is corrupted with probability $1-(1-p)^n$, which for small p can be approximated to np (we consider that a video frame is corrupted if one of its packets is lost). If we assume that a corrupted video frame is not displayed, it follows that at the same packet loss rate the video signal perceived by the receiver degrades much more than the audio.

Table VI.2 shows the percentage of video frames that are skipped at the destination in order to keep the streams synchronized. This is basically constant for both $P2$ and $P3$ protocols due to the fact that as the load increases, fewer video frames arrive at the destination and need to be processed.

VI.2 Synchronization of Shared Windows

In this section we present the experiments we performed in order to test the behavior of our shared windows synchronization algorithms in the presence of various network loads. In the previous chapter we described in detail the synchronization algorithms. Briefly, the four protocols that we have evaluated, are as follows. Protocol X1 does not perform any synchronization. In protocol X2, if an X request is late, we drop it if it is in class of X requests that can be dropped. If the X request is ahead, it is delayed until the corresponding audio arrives. In protocol X3, no X request is dropped. However, if the X windows stream is consistently behind audio, then the X client is delayed. Protocol X4 combines techniques used in protocols X2 and X3.

Table VI.4: Evaluation of the asynchrony between audio and X windows in the presence of heavy network loads [number of audio frames].

Protocol	Network load [Mbps]	Medium value	Standard deviation	Variance	Skew out of range
X1 (do nothing)	6	-16.54	12.18	148.54	71.12%
	7	-17.25	13.39	179.40	73.80%
	8	-23.49	16.26	264.47	77.77%
X2 (skip X requests)	6	-14.04	10.81	116.95	60.15%
	7	-14.70	9.58	91.78	63.63%
	8	-13.85	11.45	131.29	62.03%
X3 (delay X requests)	6	-8.41	12.34	152.50	31.48%
	7	-14.09	18.334	336.13	41.95%
	8	-9.45	13.39	179.54	34.17%
X4 (skip/delay X requests)	6	-4.12	9.43	88.95	11.37%
	7	-4.18	7.90	62.50	11.53%
	8	-5.64	11.09	122.99	18.53%

VI.2.1 Experiment Description

For each synchronization protocol we put on the network loads varying from 1 Mbps to 8 Mbs, increasing the load by 1 Mbps in each experiment. We stopped at 8 Mbps, as for higher loads, we were getting many NFS errors and the system basically stoped functioning. As we did not see any difference in performance for loads smaller than 6 Mbps, we present here the results we obtained in the case of 6, 7 and 8 Mbps loads.

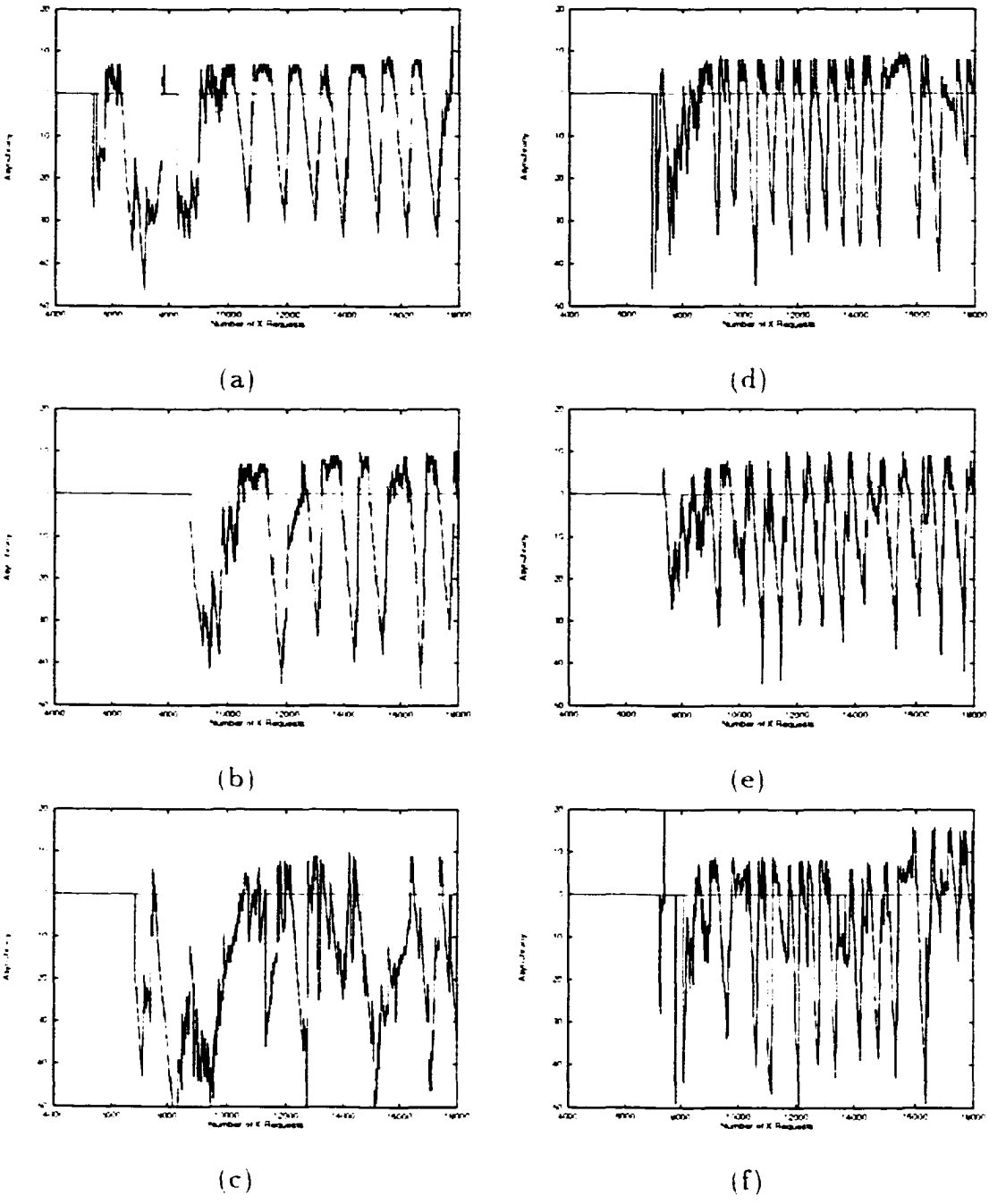


Figure VI.6: Variation of the skew between audio and the X windows stream with protocol X1 ((a), (b), (c)) and with protocol X2 ((d), (e), (f)) when a load of (a) and (d) 6 Mbps, (b)and (e) 7 Mbps, (c) and (f) 8 Mbps was put on the network.

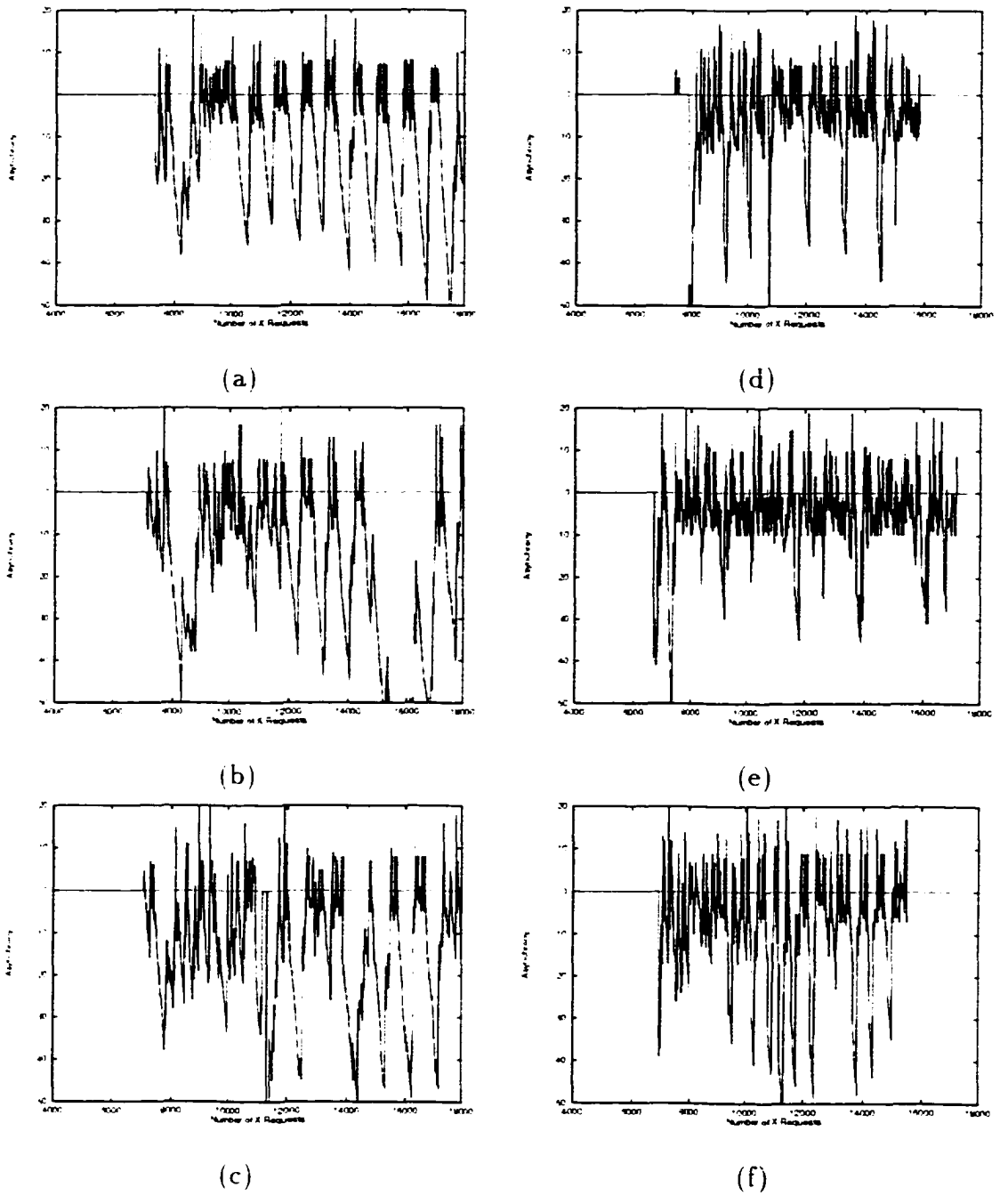


Figure VI.7: Variation of the skew between audio and the X windows stream with protocol X3 ((a), (b), (c)) and with protocol X4 ((d), (e), (f)) when a load of (a) and (d) 6 Mbps, (b)and (e) 7 Mbps, (c) and (f) 8 Mbps was put on the network.

VI.2.2 Results and Evaluation

In each experiment, we measured the skew between audio and X-windows before an X-windows packet was sent to the X server. Figures VI.6 and VI.7 show the variation of the skew. Table VI.4 presents the medium value, the standard deviation, the variance of the skew and the percentage of skews that are out of range. Ideally the asynchrony between audio and X windows should be within [-8, 12] audio frames, which corresponds to (-500, 750) ms. A negative skew means audio is ahead of X windows. A positive skew means audio is behind X windows.

With protocol X1, for a 6 Mbps load, the average asynchrony between audio and X windows was -16.54 (1058 ms). In addition, in 71.02 % of the cases the skews are larger than the maximum accepted values. This is due to the large display time of some X requests, which makes the X-windows stream to consistently lag behind audio. The presentation is annoying and the skew is visible to the user. With protocol X2, the average skew decreases to -14.04 (898.56 ms). The number of skews that are out of range decreases to 60.15 %. Although the streams are better synchronized, the image quality degrades because some X requests are dropped. With protocol X3, the average asynchrony has decreased to -8.41 (538.24 ms). The number of skews that are out of range is now 31.48 %. The best solution proved to be protocol X4 which basically combines protocols X2 and X3 (it skips the X requests and it delays the X client). As a result, the average skew is around -4.12 (263.68 ms) and the number of skews out of sync is 11.37 %.

The average asynchrony, variance and standard deviation are surprisingly stable as the network load increases. We attribute this to the fact that the increase in the load introduces similar delays to both X windows and audio packets.

VI.3 Summary

In this chapter we have evaluated our synchronization protocols for audio, video and X-windows. First, we have presented the results for the four protocols for lip-synchronization, studied for 640 × 480 pixels windows. The best performance in terms of image quality and lip-synchronization was obtained with P4, the protocol which does not drop any video frame that is late, but delays audio if it is a trend for video to be behind. For 8 Mbps loads, there is no out-of-range skew, while for 8.75 Mbps loads, 11.11% of the skews are out of range. Protocols P2 (which drops video frames that are late) and P3 (which drops video frames that are late, while also delaying audio), keep audio and video synchronized, but do not ensure a good video image.

For the synchronization of the X-windows stream with audio, the best performance was obtained with protocol X4 (drop X requests and delay the X client if the asynchrony is persistent). An average of 11.37 % of the skews are out-of-sync in case of 6 Mbps loads and 18.53 % in the case of 8 Mbps loads. Only delaying the X client (protocol X3), or only dropping X requests (protocol X2) proved not be enough to keep the streams synchronized.

Chapter VII

Results and Conclusions

"On the mountains of truth you can never climb in vain:
either you will reach a point higher up today, or you
will be training your powers so that you will be able
to climb higher tomorrow."

Friedrich Nietzsche

Multimedia synchronization is one of the key technologies for the successful delivery of distributed multimedia applications. In this thesis, we have proposed a set of algorithms that achieve the synchronization of audio, video and the X-windows streams in a distributed, collaborative multimedia application. Experimental results show that our algorithms outperform the previous algorithms in the presence of both network and hosts load variations. In this chapter we describe how we have achieved the thesis objectives presented in Chapter I. We also propose directions for future work.

VII.1 Media Synchronization Specification

While most of the existing solutions for the temporal synchronization problem take into account the network load, they largely ignore the effect of workstation load. For this reason, we started our research by studying how the workstation load variation

affects the stream synchronization.

To satisfy audio and video time constraints, multimedia processes should be scheduled periodically. This avoids audio and video device drivers queue overflow and provides a correct synchronization specification. The first question we tried to answer was whether the real-time capabilities of the current general purpose operating systems are sufficient. While in many situations the answer is yes, there are cases such as high X windows interaction, in which multimedia processes fail to be scheduled at regular time intervals.

As real-time does not eliminate the time variability when scheduling multimedia processes, we developed a new mechanism to provide a correct synchronization specification. For audio/video, we associate to each packet a sequence number based on (1) the difference between the last two scheduling times of the audio/video process, (2) the period of the stream, and (3) the number of buffers in the device driver queue. In the case of the X windows stream we simply timestamp the packet with the time when the packet arrives at the data sharing process (*xtv*). The is because from our measurements it turned out that the propagation time of an X request from the X client to *xtv* is significantly smaller than the tolerable asynchrony between audio and X windows and therefore it can be neglected.

VII.2 Media Display Time

The workstation load variation affects not only the correctness of the synchronization specification, but also the display time of media units. Two media units which are simultaneously *sent* to their presentation devices will *play* simultaneously only if their display times are identical. This is rarely the case. While audio has basically negligible display time, video has a fairly large display time, depending on the size and depth of the window.

The display time of media units is an issue that has been generally ignored by existing solutions. Based on experimental results, we also neglect the display time of an audio frame. However, to estimate the display time of a video frame we use an exponential averaging relation that adds the previous measured value (with 0.75 weight) to the previous measured value (with 0.25 weight).

For the X windows stream we have conducted experiments to see how long it takes to the X server to process each of the 127 types of X requests. This experiments confirmed the intuition that the X requests that update windows (e.g., *PutImage*) have a fairly large display time. However, to estimate the display time of an X windows packet is basically impossible, as the display time varies so much with the parameters of the request. In this situation, our synchronization algorithms ignore the display time of X windows, but apply corrections (drop X requests and delay the X client) such that within a short interval, the streams are in sync again.

VII.3 Synchronization Condition

After studying the effect of workstation load variation on the temporal synchronization problem we have studied the synchronization conditions widely used in literature (see Chapter II). Among these, we note that the conditions based on sequence numbers and synchronization points require the streams to have the same period, or a period that is a common divisor. On the other hand, the conditions based on timestamps waste valuable network bandwidth (see Chapter I for a numerical example). To address the above problems, we proposed a novel synchronization condition based on sequence numbers that can handle streams with arbitrary periods.

VII.4 Lip-Synchronization

The requirement of a synchronization mechanism between audio and video is a well

determined issue. There are numerous solutions suggested in literature for this problem (see Chapter II). Since the topic of our research is the synchronization of audio, video and X windows, we have implemented and tested first, the classical "drop-delay video" lip-synchronization algorithm. This approach proved to be inadequate for large window sizes, where the display time of a video frames is fairly large. For this reason, we have investigated two solutions. In the first one, we augment the classical "drop-delay video" solution, by delaying the audio stream whenever there is a trend of video frames being late. With this approach the two streams are synchronized, but the image freezes sometimes because of the dropped video frames. The second solution is similar with the first one, with the difference that no video frame is dropped. This solution proved to provide a synchronized presentation and a good image quality in the presence of hosts and network load variation.

VII.5 Synchronization of the Shared Windows Stream

The shared windows stream poses additional problems to the temporal synchronization. This is mainly because, unlike video and audio, the X windows is an aperiodic statefull stream that has a history and randomly dropping X requests can make the application to crash. To integrate the shared windows stream, we have proposed a mechanism that increases the number of corrections applied to the system, depending on the magnitude of the asynchrony. The first correction is to drop X requests, if this is possible. We have experimentally determined that 47 out of 127 X requests can be dropped. If this is not sufficient to get the streams back in sync, we delay the X client. This solution proved to work well in the case of various host and network loads, as well as in the case of high user interaction with the X client.

VII.6 Extension to a Distributed System

After designing the synchronization algorithms for audio, video and X windows we wanted to extend our solution to a distributed system. Media synchronization in a distributed system poses two additional issues: (1) to extract the synchronization information from mixed audio streams, and (2) to provide a global clock for all workstations.

In Chapter VI, we illustrate the first issue in the context of multiple users that speak simultaneously, and propose a solution to address it. To achieve a common time in a distributed system, we propose a statistical averaging technique which requests the starting times from the other members in the group. Our algorithm is totally decentralized in the sense that it does not assume a master workstation that keeps the reference time. As a result our algorithm is both efficient and robust.

VII.7 Future Work

Our algorithms achieve fine-grain synchronization of audio, video (CellB compressed) and shared windows, in collaborative environments that are subject to timing variability. As a future work it would be interesting to study the behavior of our algorithms when other compression techniques, like MPEG, H.261 or H.263, are used. A related question would be to determine which encoding scheme works best with audio and X windows streams. Another research direction would be to extend our algorithms to work in applications that provide VCR facilities. In this case, streams need to be played forward/backward, paused and resumed which requires buffers control both at the server and client sides.

VII.8 Impact of Contribution

The contribution of our work is the following. First, it demonstrates that not only

the network, but also the workstation load has to be considered by a correct and complete temporal synchronization solution. Recognizing the importance of this issue will hopefully prompt researchers to extend their algorithms to work well in the presence of both network and workstation load variations. Second, it proves that the synchronization of the shared windows stream in a multimedia application can be achieved most of the time in a time-sharing environment. This will hopefully encourage other multimedia applications to integrate the shared windows stream, creating more versatile and powerful shared workspaces.

References

[1] H. Abdel-Wahab and M. Feit, "XTV: a Framework for sharing X window clients in remote synchronous collaboration", *Proc. TriComm '91: Communications for Distributed Applications & Systems*, New York, pp. 159-167, January 1991.

[2] H. Abdel-Wahab, K. Maly and E. Stoica, "Multimedia integration into a distance learning environment", *Proc. of Third International Conference on Multimedia Modelling*, Toulouse, France, pp.69-85, November 1996.

[3] N. Agarwal and S.H. Son, "Synchronization of distributed multimedia data in an application-specific manner", *Proc. ACM Multimedia '94*, San Francisco, California, pp. 141-148, October 1994.

[4] D.P. Anderson and G. Homsky, "A continuous media I/O server and its synchronization mechanism", *IEEE Computer*, Vol. 1, pp. 51-58, October, 1991.

[5] B. Bailey and J. Konstan, "NSync - a constraint based toolkit for multimedia", *Proceedings of Tcl Workshop*, Boston, Massachustetts, pp. 169-177, June 1997.

[6] S. Baqai, M. Farrukh Khan, M. Woo, S. Shinkai, A. Khokhar and A. Ghafoor, "Quality based evaluation of multimedia synchronization protocols for distributed Multimedia information systems", *IEEE Journal of Selected Areas in Communications* Vol. 14, No. 7, pp. 1388-1403, September, 1996.

[7] E. Biersak, W. Geyer and C. Bernhardt, "Intra- and inter-stream synchronization for stored multimedia streams", *Proceedings of IEEE International Conference*

on Multimedia Computing and Systems, Hiroshima, Japan, pp. 372-381, June 1996.

[8] G.S. Blair, G. Coulson, M. Papathomas, P. Robin, J.-B. Stefani, F. Horn and L. Hazard, "A programming model and system infrastructure for real time synchronization in distributed multimedia systems ", *IEEE Journal of Selected Areas in Communications*, Vol.14, No.1, pp. 249-263, January 1996.

[9] G.Blakowski, J. Hubel, U. Langrehr and M. Mulhauser, "Tool support for the synchronization and presentation of distributed multimedia ", *Computer Communications*, Vol.15, No.10, pp. 611-618, November 1992.

[10] J. Bolot and P. Hoschka, "Sound and video on the Web", *Proceedings of 5th WWW Conference*, Paris, France, pp. 154-172, May 1996.

[11] S. Cen, C. Pu, R. Staehli, C. Cowan and J. Walpole, "A distributed real-time MPEG video audio player", *Proc. of the 5th International Workshop on Network and OS Support for Digital Audio and Video*, Durham, New Hampshire, pp. 50-61, April 1995.

[12] H.-Y. Chen and J.-L. Wu , "MultiSync: a synchronization model for multimedia systems ", *IEEE Journal of Selected Areas in Communications*, Vol.14, No.1, pp.238-248, January 1996.

[13] M. Correia and P. Pinto, "Low-level multimedia synchronization algorithms on broadband networks", *ACM Multimedia '95*, San Francisco, California, pp.423-434, November 1995.

[14] J.-P. Courtiat, R. C. de Oliveira and F.R. da Costa Carmo, "Towards a new multimedia synchronization mechanism and its formal specification", *ACM Multimedia '94*, San Francisco, California, pp.133-140, October 1994.

[15] D.J. Duke, D.A. Duce, I. Herman and G. Faconti, " Specifying the PREMO synchronization objects. ERCIM Technical Report, ERCIM-01/97-R048, January 1997.

[16] A. Eleftheriadis, S. Pejhan and D. Anastassiou, "Algorithms and performance evaluation of the Xphone multimedia communication system", *Proc. of ACM Multimedia '93*, Anaheim, California, pp. 401-415, August 1993.

[17] D. Ferrari, "Delay jitter control scheme for packet-switching internetwoks", *Computer Communications*, Vol.15, No.6, pp. 367-373, July/August 1992.

[18] K. Fujikawa, S. Shimojo, T. Matsuura, S. Nishio and H. Miyahara "The synchronization mechanisms of multimedia information in the distributed hypermedia system harmony", Technical Report ISE-TR-93-006, Faculty of Engineering, Department of Information and Computer Science, Osaka, Japan, September 1993.

[19] Gusella R. and S. Zatti, "Tempo - a network time controller for a distributed Berkeley UNIX system", *IEEE Distributed Procesing Technical Committee Newsletter 6, NoSI-2*, pp 7-15, June 1984.

[20] M. Hodges, R. Sassnett and M. Ackerman, "Athena Muse: a construction set for multimedia applications", *IEEE Software*, Vol.6, No.1, pp. 37-43, January 1989.

[21] P. Hoepner, "Synchronizing the presentation of multimedia objects", *Computer Communications*, Vol.15, No.9, pp. 557-564, November 1992.

[22] P.Hoscha, "Synchronized multimedia integration language (SMIL) 1.0 Specification", available at http://www.w3.org/TR/REC-smil.

[23] K. Jeffay, D.L. Stone and F.D. Smith, "Transport and display mechanisms for multimedia conferencing across packet-switched networks", *Comp. Networks and ISDN Systems*, Vol.26, No.10, pp.1281-1304, July 1994.

[24] L.Lamont and N.D. Georganas. "Synchronization architecture and protocols for a multimedia news service application", *IEEE Intl. Conf. on Multimedia Computing and Systems*. Boston, Massachusetts, pp. 309-320, May 1994.

[25] C.-S. Li and Y. Ofek, "Distributed source-destination synchronization using in-band clock distribution", *IEEE Journal of Selected Areas in Communications*, Vol.14, No.1, pp.153-161, January 1996.

[26] W. Liao and V.O.K. Li, "Synchronization of distributed multimedia systems with user interactions", *Proc. of Third International Conference on Multimedia Modelling*. Toulouse, France, pp.237-252, November 1996.

[27] C.C. Lin, S.K. Chang and T. Znati, "QoS message directed adapted distributed multimedia systems", *1997 Pacific Workshop on Distributed Multimedia Systems*. Pittsburgh, Pennsylvania, pp. 186-194, July 1997.

[28] C. J. Lindblad and D. L. Tennenhouse, "The VuSystem: a programming system for compute-intensive multimedia", *IEEE Journal of Selected Areas in Communications*, Vol.14, No.7, pp.1501-1523, July 1996.

[29] T.D.C. Little and A. Grafoor, "Scheduling of bandwidth-costrained multimedia traffic", *Computer Communications*, Vol.15, No.6, pp. 381-387, August 1992.

[30] T.D.C. Little, "A framework for synchronous delivery of time-dependent multimedia data", *Multimedia Systems*, Vol.1, No.2, pp. 87-94, 1993.

[31] K. Maly, H. Abdel-Wahab, R. Mukkamala, A. Gupta, A. Prabhu, H. Syed and C.S. Vemuru, "Mosaic + XTV = CoReview", *Proceedings of 3rd International WWW Conference*, Darmstadt, Germany, pp. 234-265, April 1994.

[32] K. Maly, H. Abdel-Wahab, C.M. Overstreet, C. Wild, A. Gupta, A. Youssef, E. Stoica and E. Al-Shaer, "Interactive distance learning over Intranets", *IEEE*

Journal of Internet Computing, Vol. 1, No. 1, pp. 60-71, February 1997.

[33] N. Manohar and A. Prakash, "Dealing with synchronization and timing variability in the playback of session recordings", *Proc. of ACM Multimedia '95*, San Francisco, California, pp. 45-56, November 1995.

[34] N. Manohar and A. Prakash, "Tool coordination and media integration on asynchronously shared computer supported workspaces", Technical Report CSE-TR-284-96, Department of Electrical Engineering and Computer Science University of Michigan at Ann Arbor, February 1996. URL page: http://www.eecs.umich.edu/~nelsonr/postcript-docs/techrpt96.ps.

[35] A. Mathur and A. Prakash, "Protocols for integrated audio and shared windows in collaborative systems", *Proceedings, ACM Multimedia '94*, San Francisco, California, pp. 381-388, October 1994.

[36] D.L. Mills, "Network time protocol (Version 3) specification, implementation and analysis", DARPA Network Working Group Report RFC-1305, University of Delaware, March 1992.

[37] S. Minneman, Steve Harrison, Bill Janssen, Gordon Kurtenbach, Thomas Moran, Ian Smith and Bill van Melle, "A confederation of tools for capturing and accessing collborative activity", *Proceedings, ACM Multimedia '95*, San Francisco, California, pp. 523-533, November 1995.

[38] S. Mullender, "*Distributed Systems*", ACM Press, 1993.

[39] Network Working Group, "RTP:A transport protocol for real-time applications", January 1996, available at ftp://ftp.ds.internic.net/rfc/rfc1889.txt.

[40] C. Nicolau, "An architecture for real time multimedia communication systems", *IEEE Journal on Selected Areas in Communications*, Vol. 8, No.3, pp. 391-400,

April 1990.

[41] J. Nieh, J.G. Hanko, J.D. Northcutt and G. A. Wall, "SVR4 UNIX scheduler unacceptable for multimedia applications", *Proceedings 4th International Workshop on Network and OS Support for Digital Audio and Video*, Lancaster, United Kingdom, pp. 192-203, November 1993.

[42] P. Owezarski and M. Diaz, "Models for enforcing multimedia synchronization in visioconference applications", *Proc. of the Third International Conference on Multimedia Modelling*, Toulouse, France, pp.85-100, November 1996.

[43] L. Qio and K. Nahrtedt, "Lip synchonization within an adaptive VOD system" *Proc. of International Conference on Multimedia Computing and Networking* , San Jose, California, pp. 206-215, February 1997.

[44] S.V. Raghavan, B. Prabhakaran and S.K. Tripathi, "Synchronization representation and traffic source modelling in orchestrated presentations", *ACM Multimedia '96 Conference*, Boston, Massachussetts, pp. 562-579, November 1996.

[45] P.V. Rangan, S. Ramanathan, H.M. Vin and T. Kaeppner, "Techniques for multimedia synchronization in network file systems", *Computer Communications Journal*, March 1993, pp. 1203-1217.

[46] O'Reilly & Associates, Inc, "X Protocol reference manual", O'Reilley Press, Vol.0, June 1993.

[47] W. Rosenberry, D. Kenney and G. Fisher, "Understanding DCE", *Annales des Telecommunications*, October 1996.

[48] K. Rothermel and T. Helbig, "Clock hierarchies: An abstraction for grouping and controlling media streams", *IEEE Journal of Selected Areas in Communications*, Vol.14, No.1, pp.174-184, January 1996.

[49] L.A. Rowe and B.C. Smith, "A continuous media player", *Proc. of the 3rd International Workshop on Network and OS Support for Digital Audio and Video*, San Diego, California, pp. 101-116, November 1992.

[50] D. Rubine, R.B. Dannenberg and D.B. Anderson, "Low latency interraction through choice-points, buffering and cuts in Tactus", *Proceedings of the International Conference on Multimedia Computing and Systems*, Los Alamitos, California, pp.224-233, May 1994.

[51] L. Rutledge and J. van Ossenbruggen, L. Hardman and D. Bulterman, "A framework for generating adaptable hypermedia documents", ACM Multimedia 97, Seattle, Washington, pp. 105-135, May 1997.

[52] B.K. Smith, J.D. Northcutt and M.S. Lam, "A method and apparatus for measuring media synchronization ", *Proceedings 5th International Workshop on Network and OS Support for Digital Audio and Video*, Durham, New Hampshire, pp.203-214, April 1995.

[53] R. Steinmetz, "Synchronization properties in multimedia systems" , *IEEE Journal of Selected Areas in Communications*, Vol.8, No.3, pp.401-412, April 1990.

[54] R. Steinmetz and K. Nahrstedt, *Multimedia: computing, communications & applications*, Prentice-Hall, 1995.

[55] E. Stoica, H. Abdel-Wahab and K. Maly, "Application embedded algorithms for multiple streams synchronization in distributed multimedia systems", *Proceedings of CS&I'97: Third International Conference on Computer Science & Informatics* Durham, New Hampshire, pp.207-227, March 1997.

[56] E. Stoica, H. Abdel-Wahab and K. Maly, "Synchronization of multimedia streams in distributed environments", *Proceedings of IEEE International Con-*

ference on Multimedia Computing and Systems Ottawa, Canada, pp.301-316, June 1997.

[57] E. Stoica, H. Abdel-Wahab and K. Maly, "Synchronization algorithms for the playback of multiple distributed streams", *Proceedings of IEEE Fourth International Conference on Multimedia Modelling* Singapore,pp. 627-636, November 1997.

[58] D. Stone and K. Jeffay, "An empirical study of delay jitter management policies", *Multimedia Systems*, Vol.2, No.6, pp.267-279, January 1995.

[59] Sun Microsystems, Inc, "SunVideo 1.0 user's guide", October 1993.

[60] J.P. Thomas, "Pseudo-tree data structure for content-based composition and synchronization of multimedia presentation", *Proc. of Third International Conference on Multimedia Modelling*, Toulouse, France, pp.253-268, November 1996.

[61] U.S. Department of Commerce, "Automated computer time service (ACTS)", NBS Research Material 8101, 1981.

[62] U. Vahalia, "UNIX internals. The new frontiers", *Prentice Hall*, 1996.

[63] D. K. Y. Yau and Simon S. Lam, " Adaptive rate-controlled scheduling for multimedia applications", *Proceedings of ACM Multimedia '96*, Boston, Massachusetts, pp. 129-140, November 1996.

Appendix A

Classification of X requests

Table A.1: X Requests that crash the X client if dropped.

Code	Description
a) Create resources	
1	CreateWindow
45	OpenFont
53	CreatePixmap
55	CreateGC
57	CopyGC
62	CopyAtrea
63	CopyPlane
78	CreateColormap
80	CopyColormapAndFree
93	CreateCursor
94	CreateGlyphCursor

Table A.2: X Requests that crash the X client if dropped (cont.)

Code	Description
b) Window manipulation by the window manager	
7	ReparentWindow
12	ConfigureWindow
c) Change resources characteristics	
2	ChangeWindow Attributes
18	ChangeProperty
24	ConvertSelection
30	ChangeActivePointerGrab
56	ChangeGC
114	RotateProperties
d) Keyboard and Pointer	
28	GrabButton
33	GrabKey

Table A.3: X Requests that freeze the X client if dropped (queries).

Code	Description	Code	Description
3	GetWindowAttributes	52	GetFontPath
14	GetGeometry	73	GetImage
15	QueryTree	83	ListInstalledColormaps
16	InternAtom	84	AllocColor
17	GetAtomName	85	AllocNamedColor
20	GetProperty	86	AllocColorCells
21	ListProperties	87	AllocColorPlanes
23	GetSelectionOwner	91	QueryColors
26	GrabPointer	92	LookupColor
31	GrabKeyboard	97	QueryBestSize
35	AllowEvents	98	QueryExtenssion
36	GrabServer	99	ListExtenssions
38	QueryPointer	101	GetKeyboardMapping
39	GetMotionEvents	103	GetKeyboardControl
40	TranslateCoordinates	106	GetPointerControl

Table A.4: X Requests that freeze the X client if dropped (cont.)

Code	Description	Code	Description
42	SetInputFocus	108	GetScreenSaver
43	GetInputFocus	110	ListHosts
44	QueryKeymap	116	SetPointerMapping
47	QueryFont	117	GetPointerMapping
48	QueryTextExtensions	118	SetModifierMapping
49	ListFonts	119	GetModifierMapping
50	ListFontWithInfo		

Table A.5: X Requests that affect other X clients if dropped.

Code	Description	Effect on other X clients
25	SendEvent	an X client may be blocked waiting for the event
27	UngrabPointer	user cannot point in any other window
29	UngrabButton	user cannot use the button in another window
32	UngrabKeyboard	user cannot type in other window
34	UngrabKey	user cannot use the key in another window
37	UngrabServer	X server cannot process other connections
109	ChangeHosts	a host may not be able to connect to local server
111	SetAccessControl	enable/disable access control list
115	ForceScreenSaver	reset/activate screen saving

Table A.6: X Requests that can be safely dropped.

Code	Description
a) Destroy resources	
4	DestroyWindow
5	DestroySubWindows
19	DeleteProperty
46	CloseFont
54	FreePixmap
60	FreeGC
79	FreeColormap
82	UninstallColormap
88	FreeColors
95	FreeCursor
107	SetScreenSaver
112	SetCloseDownMode
113	KillClient
b) Window manipulation by the X client	
8	MapWindow
9	MapSubWindows
10	UnmapWindow
11	UnmapSubWindows
13	CirculateWindow

Table A.7: X Requests that can be safely dropped (cont.)

Code	Description
c) Draw graphics	
61	ClearArea
64	PolyPoint
65	PolyLine
66	PolySegment
67	PolyRectangle
68	PolyArc
69	FillPoly
70	PolyFillRectangle
71	PolyFillArc
d) Put text	
74	PolyText8
75	PolyText16
76	ImageText8
77	ImgeText16
e) Put image	
72	PutImage

Table A.8: X Requests that can be safely dropped (cont.)

Code	Description
f) Keyboard and Pointer	
41	WarpPointer
96	RecolorCursor
100	ChangeKeyboardMapping
102	ChangeKeyboardControl
104	Bell
105	ChangePointerControl
g) Miscellaneous	
6	ChangeSaveSet
22	SetSelectionOwner
51	SetFontPath
58	SetDashes
59	SetClipRectangles
81	InstallColormap
89	StoreColors
90	StoredNamedColors
107	SetScreenSaver

Vita

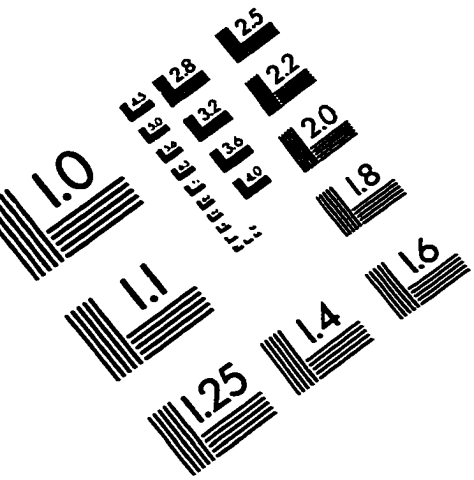
Emilia Stoica was born in Bucharest, Romania on May 23, 1966. She received her Master degree in Computer Science and Engineering from Polytechnical Institute of Bucharest, Romania, in June 1989. She worked as a Software Engineer for IIRUC, Bucharest, Romania, from October 1989 until June 1993. From June 1993 until January 1994, she worked as a Systems Engineer for International Computer Limited (ICL), Bucharest, Romania headquarters. In May 1994, she started working on her Ph.D Degree in Computer Science at Old Dominion University, Norfolk, Virginia. Mrs. Stoica is currently Systems Designer at the Research Department of Claritech Corporation, Pittsburgh, PA.

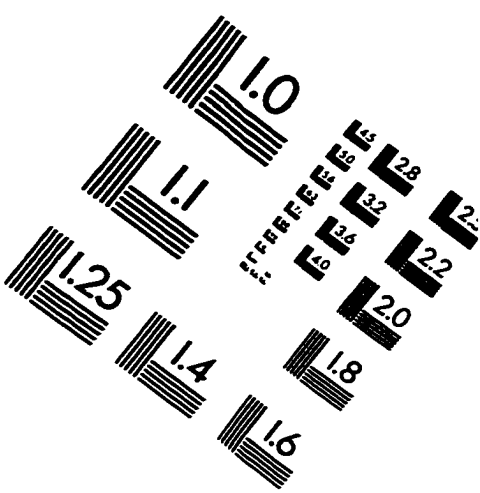
Permanent address: Department of Computer Science
Old Dominion University
Norfolk, VA 23529
USA

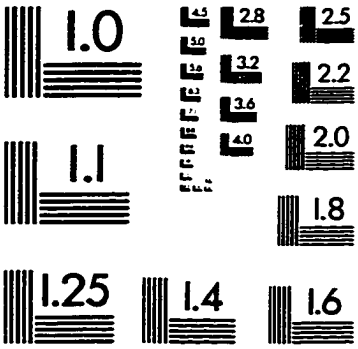
This dissertation was typeset with LaTeX[†] by the author.

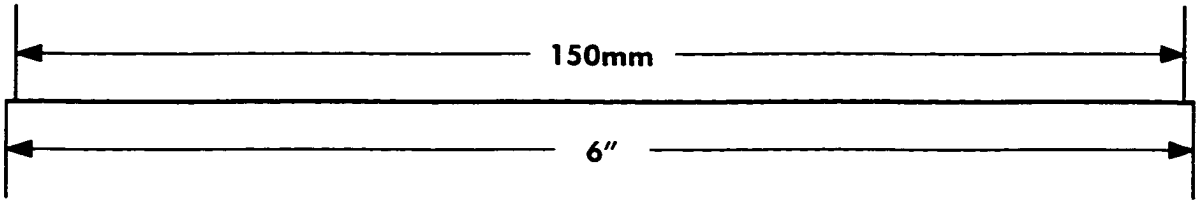
[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.

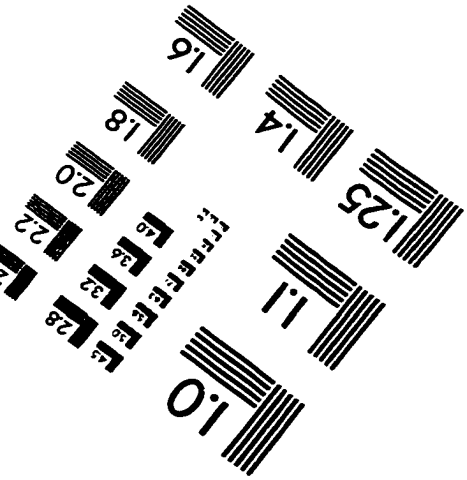
IMAGE EVALUATION TEST TARGET (QA-3)

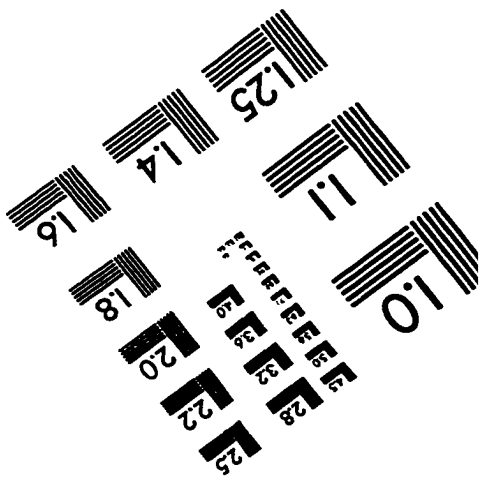












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989