# Dealing with Synchronization and Timing Variability in the Playback of Interactive Session Recordings

*Nelson R. Manohar and Atul Prakash*
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122 USA.
email: {nelsonr,aprakash}@eecs.umich.edu

## ABSTRACT

In this paper, we describe scheduling and synchronization support for a novel multimedia document, referred to as a *session object*. The session object captures a voice-annotated, interactive session with an application — it contains audio and window streams. This paper addresses media scheduling and synchronization issues for the support of faithful replay of session objects when subject to timing variability at the replay workstation. The replay is supported by an adaptive scheduling algorithm. The algorithm preserves relative inter-stream synchronization between window and audio streams. Run-time temporal deformations are applied over the schedule of the window stream. We show that the inter-stream asynchrony floats under statistical control as a function of the scheduling interval. The mechanisms could be generalized to the replay of streams that are subject to timing variability. Our object-oriented toolkit, ReplayKit, enables an application to become replay-aware through access to session capture and replay functionality.

## KEYWORDS

Media integration and synchronization, collaboration environments, and session capture and replay.

## INTRODUCTION

In synchronous collaboration, users of a multi-user application first find a common time and then work in a WYSIWIS (What You See Is What I See) collaborative session. However, a synchronous mode of collaboration can often be too imposing on the schedule of the participants. It requires that users be able to find a common time to work together but, in many cases, that is not easy. In [24, 25], we presented the WYSNIWIST (*What You See Now, Is What I Saw Then*) paradigm for asynchronous collaboration that allows users to record and replay an interactive session with an application.

The paradigm introduced an associated data artifact, the *session object*, used to capture an interactive session. Figure 1 shows a high level view of the capture and replay of an interactive session with an application. During the capture of the session, user interactions with the application, audio annotations, and resource references (e.g., fonts, files, environment) are recorded into a *session object* (Fig. 1a). The replay of the session uses the data stored in the session object to simulate the original session (Fig. 1b).

Our replay approach is application-dependent. During replay, input events are re-executed (as opposed to a passive replay form such as a series of screen dumps). The re-execution approach to session replay exhibits benefits that are of particular interest in collaborative work:

- If a participant misses a collaborative session, our approach allows the participant to replay the session, catch up with the team, and if desired, continue further work.
- Because input events are recorded (as opposed to recording display updates or even screen dumps), session objects are typically small in size and thus easier to exchange among collaborators.
- Because the application is re-executing the original session (as opposed to re-displaying screen dumps of a session), the highest possible fidelity of replay is achieved, which for some domains may be essential — consider the medical domain discussed below.

The following examples, being pursued as part of our research, illustrate the needs and benefits for session objects in asynchronous collaboration scenarios:

**UARC:** The ReplayKit research was originally motivated by the UARC project, (see Fig. 2), a collaboratory experiment among domain scientists in a wide-area network [8]. The domain of research among the scientists is space science. Because domain scientists often work from different time-zones and it is not known a-priori when interesting phenomena will be observed, providing support for session capture, annotation, replay, and exchange should facilitate collaborative work among scientists.

**MedRad:** As part of an NSF project, we plan to support the type of collaboration that occurs between a radiologist and a doctor over radiographs to diagnose a patient's medical problem. We would like a radiologist or a doctor to be able to record a session in which they are interacting with one or more images, pointing to specific areas of interest, using audio to explain their understanding or raise questions about regions of interest in the images, and adding text or graphical annotations. They can collaborate by exchanging such recordings. Such digital, high resolution session recordings will not only help to capture radiologists'

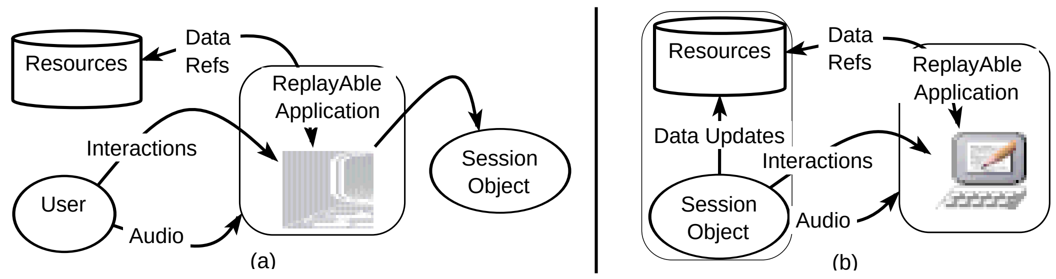

Figure 1. Capture and replay of an interactive session with a REPLAYABLE application. During the capture of the session, user interactions, audio-based annotations, and resource references are encapsulated into a persistent, transportable, session object. During the replay, the session object makes these accessible to the application.

Figure 2. Snapshot of the interface presented by the multi-user UARC software to one of the users. During a session, experts navigate through many views and settings. Replay of these sessions is of interest to them.

> *diagnostic conclusions*, but also their *diagnostic process*. This is important because in many instances how the diagnosis was reached is as important as the diagnosis itself.

The approach, however, requires addressing issues such as capture of application state, assumptions of determinism, and *timing-variability* during re-execution of events (due to performance differences in replay workstations, varying load conditions, timer inaccuracies, etc.). The focus of this paper is on addressing timing variability issues.

Session objects are used for (asynchronous) collaborative work. The basic collaboration mechanisms allow a participant who misses a session with an application to catch up (and if desired, continue further work) on the activities that occurred during the session. The session object captures a voice-annotated, interactive session with an application — it contains audio and window streams, modeled as time-based discrete streams. Session objects are replayed by re-executing these streams.

Session objects are transportable multimedia objects. The re-execution of events from a stream is subject to timing variability. During replay, voice annotations (i.e., the audio stream) must maintain relative timing *wrt* user interactions (i.e., the window stream). That is, synchronization must be performed *wrt* the relative progress of a stream and not *wrt* the progress of schedule time. The window stream is a stateful, aperiodical, discrete, and asynchronous stream. The audio stream is a stateless, periodical, continuous, and synchronous stream. Adaptive strategies for integrated replay of stateless, periodical and continuous media can not efficiently address requirements of the window stream (e.g., dropping/duplicating frames cannot be usually allowed). This paper discusses media scheduling and synchronization issues for the support of faithful replay of session objects.

The rest of the paper is organized as follows. First, we present the goals and requirements of session objects. Then, we discuss related work. Next, we discuss modeling and design issues. Then, we compare the performance of several synchronization protocols and analyze our findings. Next, we describe our implementation. Finally, we present our concluding remarks. Also, in Appendix A, we further discuss our experimental setup, and in Appendix B, we formally specify our adaptive scheduling algorithm.

## GOALS AND REQUIREMENTS

The UARC and the MEDRAD domains illustrate the need for transportable objects, suited for replay across similar workstations. The support of these domains imposes the following requirements over our approach to session replay:

**R1:** Support transportable and faithful replay of a session with an application's workspace. Synchronizing a session's streams requires preserving the relative progress of its streams. In particular, it requires compensating for timing variability introduced by different load conditions and workstations.

**R2:** Reduce reliance on hardware and operating systems support, so as to reach a broad collaboration base. In particular, it requires compensating for timing variability introduced by large overhead timing services.

**R3:** Support of interactions with the interactive workspace of the session being replayed.

This paper focuses on timing variability issues in the re-execution of discrete, functional streams in the replay workstation. Our research focuses on:

- integration of stateful, aperiodical, discrete, asynchronous media *wrt* synchronous media.

- management of timing variability due to prefetch, schedule, execution, and synchronization of multiple streams in a single workstation.
- scheduling subject to large overhead timing services.
- desirability for application layer control of the asynchrony.

Our contributions are twofold. First, to **address timing variability**, a new algorithm for run-time, adaptive scheduling for the support of integrated media replay is proposed. The algorithm periodically adjusts, if necessary, the run-time schedule of a stream, to reverse trends in inter-stream asynchrony. Adjustments are made to the scheduler of a stream and not to a global session scheduler. We show that inter-stream asynchrony floats under statistical control as a function of the scheduling interval.

Second, **our object-oriented toolkit**, ReplayKit, provides the Replayable object class. The ReplayKit toolkit extends session capture and replay functionality into applications. Through subclassing, an application inherits session capture and replay behavior as well as transparent access to our infrastructure. The toolkit allows applications to: (1) re-execute window events (e.g., gesturing, typing, moving windows), (2) record and replay voice-annotations, (3) provide synchronized replay of these streams and (4) to replay selected streams. In addition, an API for capture and replay of data streams is provided.

## RELATED WORK

Our research in the replay of interactive session recordings relates to research in (1) the replay of application workspaces, (2) the replay of stored multimedia, and (3) adaptive scheduling for the support of replay.

### The Replay of Application Workspaces

The replay of a user session with an application workspace has collaborative value [1, 25]. The following are approaches to replay a user session with an application's workspace.

**Screen recorders**, such as WatchMe (for NeXTs) and QuickTime Conferencing (for the Macs), represent an application-independent approach to session capture and replay. Capture of a session occurs at the screen level, intercepting and recording screen updates (or even screen dumps). Consequently, the replay of a session reproduces only the external look of the workstation's screen. Since interaction with the application's workspace is not possible, its suitability for asynchronous collaboration work is also reduced.

Systems such as SharedX, Xtv [1], and Ceced [10] also represent an application-independent approach that can support session capture and replay. Capture of a session is done by intercepting events sent by an application to a collaboration-aware server [7]. Although this approach, in principle, allows replay of unmodified applications, interacting with the session's application workspace is not possible. Furthermore, currently, these systems do not offer fine-grained audio synchronization support. The ideas presented in this paper can be used to extend these systems to include synchronized audio support.

**Toolkits** and **application-specific prototypes** represent application-dependent approaches to session capture and replay. The Scoot toolkit [9] proposes two approaches to capture and replay of interactive sessions. The first approach logs periodical application snapshots. This choice results in coarse replay progress feedback and limited synchronization precision. The second approach logs method invocations to log-aware objects. However, synchronization of audio *wrt* the replay of such an asynchronous log stream is not addressed. Finally, the prototypes found in [17, 26] capture and replay both audio and window streams. However, their synchronization requirements are simplified since only MouseMoved events, (to draw and move the pen), are replayed. This removes significant timing variability from the replay of the window stream.

A **media server** approach is taken by the Tactus system [11, 12]. Tactus is actually composed of an application-independent media server and an application-dependent toolkit. There are some important differences. First, the Tactus server assumes the use of reliable timing services at the scheduler. Our work, on the other hand, targets timing services as a source of timing variability. Second, Tactus's mechanisms to deal with schedule departures — (1) pre-computation and (2) throttle control — do not suit our domain requirements. Although pre-computation works well for continuous media presentations (i.e., synchronous and stateless media), it not clear whether it extends to the replay of sessions containing both asynchronous and stateful media. Tactus's throttle control implements two speeds: normal speed (used for replay) and maximum speed (used to catch-up). This increases the risk of abrupt updates to the relative progress of one or more streams (which in our domain is undesirable). On the other hand, we make use of smoothed throttle controls over the relative replay rate of a stream.

### The Replay of Stored Multimedia

Our work relates to research in the replay of stored multimedia. There are two approaches to the replay of stored media: network-based replay and workstation-based replay. However, there are important differences, explained as follows.

Research in network-based continuous media players, such as [3, 23, 30, 31], addresses **different sources of overheads**. The replay of stored media has three basic sources of overhead: (1) fetch, (2) process, and (3) presentation. In network-based replay, the media server implements media access, buffering and synchronization tasks. Variability is primarily attributed to latencies in the network.

Workstation-based replay has different sources of overhead. Our approach is workstation-based replay (i.e., a multiple stream, single-site model), since collaboration is asynchronous and access to local disk(s) is preferable. In our approach, variability is due to two factors: (1) varying *latencies* in fetching events from the local disk and (2) variability, such as *timer inaccuracies* and *variability in execution time* introduced in the scheduling and re-execution of events, respectively. One can deal with varying latencies by attempting to minimize them via buffering. However, both timer inaccuracies and the variability in execution time are more difficult to deal with. Our work extends existing workstation-based replay management of stored media through integration of stateful, asynchronous media that is subject to timing variability.

Research in the management of **fetch overheads**, such as in [20, 29, 30] focus on low-level file system *extensions* (such as predictive prefetching, disk layout, optimal buffering, etc.) for the support of continuous media streams. Our work can be built of top of these and benefit from this research.

Research in the management of **scheduling overheads**, such as in [14, 28, 19] focus on low-level operating system techniques for the support of multimedia (such as bounding of scheduling latencies, use of pre-emptive deadline scheduling, use of correlated task scheduling, etc.). Our work differs from these in several points. First, these techniques assume that the media stream is composed of periodical tasks, each with the same constant duration. On the other hand, our fine-grained asynchronous media (e.g., window stream) has neither periodicity nor constant duration. Second, these techniques tend to remove interrupt costs from their formulation. On the other hand, our timing variability model is a statistical formulation to compensate for interrupt and similar costs, through indirect performance measurements.

Finally, continuous media players assume negligible **presentation overheads** for the presentation of the streams. Our domain requires fine-grained integration of both continuous and discrete media. The playback integrated media affects our choices for both scheduling and synchronization, as found in [26]. The re-execution of fine-grained, discrete, asynchronous media requires additional compensation for timing variability on the re-execution of the stream. Both fetching and scheduling overheads introduce contention to the re-execution overheads, as a result, the fine-grained synchronization of asynchronous media establishes a need for an end-to-end (i.e., fetch, schedule, and execution) solutions to the management of overheads

### Adaptive Scheduling Strategies for Replay

The scheduling of discrete events has also been addressed in **computer-based musical systems** [2]), however, there are two major differences. First, no variability is introduced by the re-execution of MIDI events on the real-time synthesizer. Second, integrated media replay is not supported. Steinmetz [32] provided early illustrations for the need for integration of synchronous and asynchronous media.

Some issues in specification and presentation of multimedia documents have been addressed in the FIREFLY system [6] and in CMIF [5]. However, the focus of their work is on specification and enforcements of **synchronization constraints** among high-level parts of a multimedia document. Synchronization at internal points among media segments is not addressed. In our work, the focus is on enforcing fine-grain synchronization at internal points among streams.

Research in multimedia network interfaces, such as [13, 18, 26, 27, 31], propose **adaptive scheduling protocols** to manage bounded network variability. Their focus is on the management of overheads up to the delivery of data to the presentation workstation. These approaches assume negligible overheads on the processing and presentation of streams by the client workstation. Since asynchronous media is particularly sensitive to timing variability, relative inter-media timing can be lost. Our approach focuses on timing variability issues at the client. As a result, our adaptive mechanisms can be used on top of these approaches.

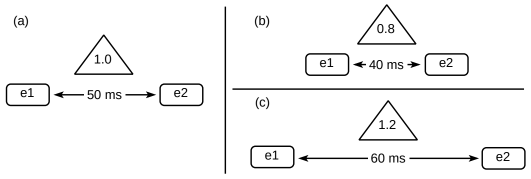

Figure 3. Intuitive look at our adaptive scheduling mechanism. Part (a) of this figure shows the inter-event delay $\Delta_{1,2} = 50ms$ between events $e_1$ and $e_2$. Our approach adapts inter-event delays (i.e., the idle time between events) to compensate for trends in asynchrony. Adjustments are made independently to slave streams. Part (b) shows a possible adjustment when this stream runs behind schedule. In this case, the inter-event delay is scaled down — the total wait or idle time observed by this stream is reduced. For example, if the original inter-event delay was $50ms$, using a compensation factor of $80\%$ decreases the wait to $40ms$. Independent of other streams, this stream is awarded a $20\%$ time credit towards its schedule. Part (c) shows a possible adjustment when this stream runs ahead of schedule. The inter-event delay is scaled up — effectively, the amount of wait or idle time observed by this stream is increased.

## MODELING AND DESIGN

In this section we discuss scheduling and synchronization issues for the support of integrated replay of window and audio streams, when subject to timing variability. We start by presenting our media model and introduce the timing variability problem.

Our proposed solution to this problem has two parts:

- a synchronization mechanism and
- an adaptive scheduling mechanism.

First, our synchronization scheme and its operation are described. Finally, our adaptive scheduling mechanism and its derived scheduling algorithm are analyzed. Figure 3 provides an intuitive look to our adaptive scheduling mechanism.

### Asynchronous Media

The prototype *currently* supports two streams, window and audio. Both window and audio streams were modeled as discrete streams, however, with clearly different tolerances.

During re-execution, events from both streams are subject to timing variability. Therefore, the time needed for re-executing an event or frame $e_i$ is modeled as:

$$t(e_i) = E(e_i) + f_o(e_i) \tag{1}$$

$E(e_i)$ represents the expected execution time for an event. $f_o(e_i)$ models variability introduced in the handling and re-execution of $e_i$. In particular, variability in the re-execution of window events must be compensated to maintain the original timing behavior *wrt* the audio stream. Variability in the re-execution of audio frames must be compensated too, to preserve continuity of playback.

What are the *sources of timing variability*? Two competing random processes introduce timing variability into the replay: the overhead functions $f_o(A)$ and $f_o(W)$, where $A$ is the audio stream and $W$ is the window stream. The $f_o(A)$ overheads have the following components:

**A1:** prefetch overheads, due to disk access of audio frames,

**A2:** scheduling overheads, due to thread overheads, preemption, timing services, context switches, etc., and

**A3:** presentation and synchronization overheads.

The $f_o(W)$ overheads have the following components:

**W1:** prefetch overheads,

**W2:** scheduling and re-execution variability due to timer inaccuracy, pre-emption, context switches, CPU availability, page faults, etc., and

**W3:** synchronization overheads due to locks, signals, preemption, context switches, CPU scheduling, etc..

The playback of a session object on a workstation requires balanced management of prefetch, schedule, and synchronization tasks and their overheads. Since these tasks compete locally for resources within a single workstation, the susceptibility of the playback session objects to timing variability is amplified. Overhead components A1 and W1 can be dealt by using buffering strategies. The management of timing variability introduced by overheads A2, A3, W2, and W3 is the focus of this paper.

### Synchronization Precision

Application layer control of the synchronization process is needed to reduce platform dependencies. A high level modeling paradigm for the replay of stream facilitates integration of the toolkit with existing applications. To support these goals, each stream is replayed by separate thread-based handlers (see Appendix A). A duality between streams and threads exists. Scheduling of stream handlers is inherited from OS thread-based scheduling primitives. Synchronization of stream handlers is inherited from OS thread-based synchronization primitives. New streams can be introduced through modular thread-based handlers. Using thread-based stream handlers, however, reduces our synchronization precision since thread models are subject to interrupts, preemption, and large system call overheads while lack peer-to-peer guaranteed scheduling time. Our synchronization precision was targeted to support about 0.1 to $1s$, (e.g., audio-annotated slide shows as quoted from [4]). Finer grain synchronization would have imposed demands requiring OS-level support [4] and thus compromising our goal for high level support (R2).

### Synchronization Operations

Under the presence of timing variability, a way to preserve the relative synchronization of streams is needed. Synchronization is based on the use of synchronization events, widely accepted in the synchronization literature [32]. A synchronization event, $s_i(lhs \leftarrow rhs)$, preserves relative timing between $lhs$ and $rhs$ streams involved in a synchronization dependency. In this notation, $rhs$ streams synchronize to the $lhs$ stream. In terms of temporal specification notations found in [21, 22], our synchronization specifically preserves the relative synchronization relationship ($e_i$ *same* $a_j$) between window and audio streams. Figure 4 shows the use of a synchronization event $s_i(A \leftarrow W)$, where $A$ is the audio stream and $W$ is the window stream.

Synchronization events are used as follows. During the capture of a session, a synchronization event is posted at a well-defined endpoint of a $lhs$ stream (e.g., the end of an audio frame) and it is then inserted into all $rhs$ streams (e.g., window), thus establishing a relative synchronization dependency. During the replay of the session, the scheduling of a synchronization event triggers an attempt at inter-stream synchronization. Inter-stream asynchrony between consecutive synchronization events is variable. Each synchronization event attempts to reset inter-stream asynchrony to zero. The asynchrony is estimated by the substraction of the observed schedule times of the latest synchronization event seen at each stream — e.g., $\epsilon_i = t(s_i(A)) - t(s_i(W))$.

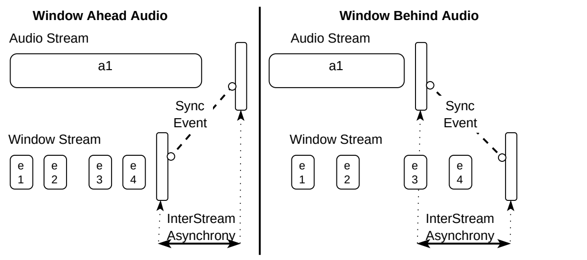

Figure 4. The two inter-stream asynchrony cases. A synchronization event, $s_i(A \leftarrow W)$, preserves relative synchronization between our $A$ (audio) and $W$ (window) streams. The inter-stream asynchrony between corresponding synchronization events is variable. It is estimated by the substraction of observed schedule times for synchronization events at each stream — e.g., $\epsilon_i = t(s_i(A)) - t(s_i(W))$.

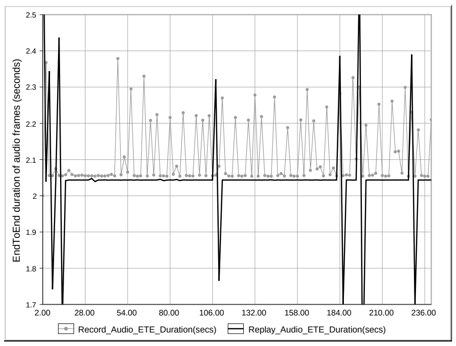

Figure 5. Overhead in record and replay of audio frames. Plots observed $t(a_i)$ duration, (as in equation 1), of audio frames during record and replay of the $250sec$ test session. Variability $f_o(A)$ normally present on the master stream is removed during replay since the master stream not longer initiates inter-stream synchronization protocols.

### Inter-stream Synchronization Mechanism

The replay of streams must be kept synchronized. Our synchronization model is based on the notion of a master and multiple slave streams, (as in [4, 15, 32]).

However, unlike these master/slave models, our approach differs in the following ways. First, synchronization is relative to the progress of the master stream (as opposed to the progress of logical time, as in the TACTUS system [11]). In our prototype, the audio stream is used as the master stream

| Protocol | Window Ahead Audio Treatment | Window Behind Audio Treatment |
|---|---|---|
| P1: *Laiseez Faire* | do nothing | do nothing |
| P2: 1 Way Blocking | wait for matching audio event. | do nothing |
| P3: 1 Way Adaptive | (1) wait for matching audio event (2) if this is a trend (to be ahead), compensate by decreasing replay speed. | (1) if this is a trend (to fall behind), compensate by increasing replay speed. |
| P4: 2 Way Blocking | (1) wait for matching audio event. | (1) if asynchrony $> T_{max}$, resync audio and window streams (audio waits). |

Table I. Specification and side-by-side comparison of discrete streams synchronization protocols. The first column lists the protocol name, followed by the protocol handling of inter-stream asynchrony. The window stream implements the treatment. $T_{max}$ refers to the maximum asynchrony departure that can be tolerated without requiring a synchronization restart.

and the window stream is its synchronizing slave.[1] We chose to synchronize window to audio because unlike the window stream, audio has stringent temporal requirements.

Second, synchronization is initiated by the slave streams (rather than by the master stream). This shift of initiator responsibility also shifts synchronization overheads $f_o(A)$, (normally present in the master stream), to overheads $f_o(W)$ on the slave streams. This shift of synchronization semantics had the following benefits: (1) reduced overheads $f_o(A)$ as seen by the master stream and (2) relaxed semantics, that facilitate relative synchronization. Figure 5 shows the effects of this scheme on the overhead $f_o()$ as seen by the master (audio) stream. Variability normally present on the master stream is reduced during replay since the master stream no longer initiates inter-stream synchronization protocols. Overall, we found this scheme to yield better audio continuity than a master-initiated synchronization scheme.

## Synchronization Protocols

A synchronizing operation can be described (as in [32]) by:

(1) the involved partner(s), (i.e., to which stream to synchronize). Our prototype implements $(A \leftarrow W)$.

(2) the type of synchronization, (i.e., whether to use a non-blocking, 1-Way or 2-Way blocking protocol).

(3) the release mechanism, (whether and how to adapt the scheduling of a stream).

In this section, we address the latter two.

Based on the use of both a master/slave model and synchronization events, we evaluated several synchronization protocols. The synchronizing operations of these protocols can be compared by examining their handling of a synchronization event, as discussed below.

When performing a synchronization check, the inter-stream asynchrony between the slave and the master stream is estimated. There are two cases of asynchrony to consider, both illustrated in Fig. 4:

- (window ahead of audio condition): the window event stream reaches its synchronization event before the audio stream (right side of Fig. 4).
- (window behind audio condition): the window event stream reaches its synchronization event after the audio stream (left side of Fig. 4).

Table I specifies these protocols in terms of their handling of these asynchrony cases. Protocol P1 and P2 relate to Gibbs' NoSync and InterruptSync synchronization modes between master and slaves found in [16]. Protocol P3 is our adaptive scheduling algorithm. Protocol P4 is a two-way, stop and wait, protocol for relative synchronization of discrete streams.

## Adaptive Scheduling Mechanism

The basic idea behind our adaptive mechanism was illustrated in Fig. 3. Next, we analyze this adaptive mechanism.

Our approach to per-stream scheduling intervals is different from the use of global (that is, across all streams) scheduling intervals. The scheduling interval of a stream contains all events between consecutive synchronization events $s_i$ and $s_{i+1}$. In general, because of the asymmetry of 1-Way protocols, a window synchronization event $(A \leftarrow W)$ can only preserve synchronization under the (window ahead audio) condition. However, under the (window behind audio) condition, nothing is done. We propose the following adaptive scheme to address the (window behind audio) condition.

In general, the duration of the scheduling interval for $n$ events in a stream, $(s_0, e_1, e_2, \cdots e_n, s_1)$, is of the form of

$$T = t(e_1) + \Delta_{1,2} + \cdots + t(e_n) + \Delta_{n,s_1} \qquad (2)$$

In a discrete stream, an inter-event delay time $\Delta_{i,i+1}$ separates any two consecutive events $e_i, e_{i+1}$.

Because re-execution time $t(e_k)$ is subject to variability $f_o()$, as defined in equation (1), the *realizable scheduling interval duration* $T$ becomes

$$T = E(e_1) + f_o(e_1) + \Delta_{1,2} + \cdots + E(e_n) + f_o(e_n) + \Delta_{n,s_1} \quad (3)$$

Ideally, a faithful replay of these events should follow the same timing observed during the recording of these events. By assuming the overheads $f_o() \rightarrow 0$, the *ideal scheduling interval duration* $T^*$ is then modeled as

$$T^* = E(e_1) + \Delta_{1,2} + \cdots + E(e_n) + \Delta_{n,s_1} \qquad (4)$$

When comparing the ideal schedule duration $T^*$ against the realizable schedule duration $T$, we obtain the *effective departure from timing faithfulness.* This is estimated by

$$\epsilon^* = T - T^* = \sum f_o(e_i) \qquad (5)$$

Our algorithm is based on the use of *time deformations* over the inter-event delay of event in a scheduling interval. Time deformations are used to better adapt $\epsilon^*$ — the timing

[1]— the window stream schedule is adjusted to preserve synchronization *wrt* audio stream's schedule.

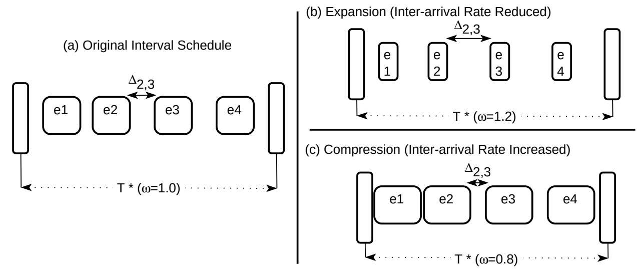
(a) Original Interval Schedule
$\Delta_{2,3}$
e1
e2
e3
e4
T * (ω=1.0)
(b) Expansion (Inter-arrival Rate Reduced)
$\Delta_{2,3}$
e 1
e 2
e 3
e 4
T * (ω=1.2)
(c) Compression (Inter-arrival Rate Increased)
$\Delta_{2,3}$
e1
e2
e3
e4
T * (ω=0.8)

Figure 6. Time deformations and $\omega$-modified scheduling intervals. Time line diagram showing potential adjustments to a scheduling interval of the window stream. Part (a) shows the original scheduling interval. Part (b) shows the effects of the expansion time-deformation: the $\omega$-modifed scheduling interval compensates with idle time a trend to faster execution. Part (c) shows the effects of the compression time-deformation: the $\omega$-modifed scheduling interval compensates with idle time a trend to slower execution.

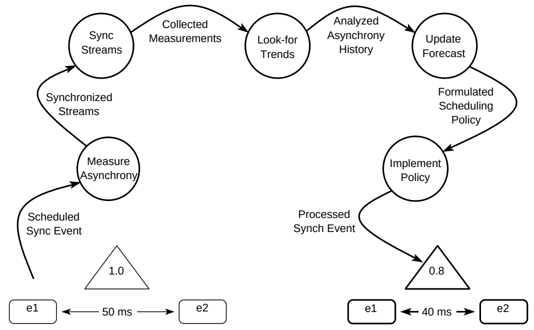

Figure 7. Overview of the per-stream adaptive scheduling process. The $\omega = 1.0$ compensation is revised to $\omega = 0.8$, resulting in 20% smaller inter-event delays for events in the next scheduling interval. A synchronization event may trigger a five phase analysis cycle to measure, synchronize, analyze, forecast, and prevent further inter-stream asynchrony.

departure — to timing variability during replay at a workstation. The basic mechanism was previously illustrated in Fig. 3. In Fig. 6, we illustrate the effect of a time deformation over the scheduling interval, when subject to timing variability trends. On each scheduling interval, a time deformation may be applied — when granted by trends in asynchrony. In such case, the time deformation is applied by scaling the inter-event delay of all events in the current scheduling interval by a factor $\omega$ — (a smoothed compensation factor, defined in Appendix B). Consequently, we obtain a *$\omega$-modified scheduling interval duration* $T_\omega$ formulated as

$$T_\omega = E(e_1)+f_o(e_1)+\omega\Delta_{1,2}+\cdots E(e_n)+f_o(e_n)+\omega\Delta_{n,s_1} \quad (6)$$

When comparing the ideal schedule duration $T^*$ against our $\omega$-modified schedule duration $T_\omega$, we obtain the *compensated departure from timing faithfulness*, $\epsilon_\omega$. This is estimated by $\epsilon_\omega = T_\omega - T^*$. Equivalently,

$$\epsilon_\omega = \sum f_o(e_i) - (1-\omega)\sum \Delta_{i,i+1} \quad (7)$$

which by equation (5) can be rewritten as

$$\epsilon_\omega = \epsilon^* - (1-\omega)\sum \Delta_{i,i+1} \quad (8)$$

This is important because now, the replay of a $\omega$-modified scheduling interval accounts for the estimated departure from timing faithfulness, $\epsilon^* = \sum f_o()$. Trends in asynchrony were modeled as departures from timing-wise faithfulness to the original schedule. A compensation factor $\omega$, determined at run-time, was used to produce compensating time deformations $(1-\omega)$. The time deformations $(1-\omega)$ are applied over the inter-event delay distribution $\Delta_{i,i+1}$ of events in the scheduling interval. These time deformations scaled up or down, (as needed), the schedule of a stream with trends in asynchrony. The schedule compensation adjustment is variable and revised, (upgraded or downgraded, as needed), on every scheduling interval. However, *asynchrony floats*, under statistical control, during the scheduling interval.

Our time deformations are different from temporal transformations, (as in [2, 15, 23, 30]). Temporal transformations scale the rate of execution of a global scheduler so as to support features such as fast forward or fast replay. Our time deformations are a mechanism to compensate inter-stream asynchrony, through variable, graded compensations, to the schedule of a slave stream.

## Stream Scheduler(s)

Our scheduling strategy consists of applying time deformations to scheduling intervals of a stream. Our $\omega$-modified scheduling intervals compensate for trends in inter-stream asynchrony. Run-time schedules are independently revised, for every discrete stream, on every scheduling interval.

Our scheduling is statistical. The behavior of the adaptive algorithm is specified as follows.[2] If the current asynchrony is large enough, the asynchrony history is examined to determine the presence of trends *wrt* $(2\sigma)$ warning and $(3\sigma)$ control limits.[3] If a trend exists, a run-time, compensated, $\omega$-modified scheduling interval for the window stream is formulated. The effects of this compensation over the scheduling interval were illustrated in Fig. 6. Part (b) shows a compensated scheduling interval for the window ahead of audio condition. Part (c) shows its effect under the window behind audio condition.

Each stream scheduler attempts to maintain statistical process control over inter-stream asynchrony between the slave stream and its master. Smoothed forecasts are used to determine statistically significant trends in asynchrony.

Each stream implements a differential time scheduler. On each scheduling interval of a stream, the scheduler dispatches an event and then, sleeps for the event's (compensated) inter-event delay time. The scheduler cycles between these two tasks (sleep and dispatch), for all events in the current scheduling interval. Note that during the scheduling interval, no timing services lookups are performed. At the end of the scheduling interval, (due to the scheduling of the synchronization event), we (1) flush any buffered window events, (2) measure the inter-stream asynchrony, and then (3) initiate

[2]— formally specified in Appendix B.

[3]— where $\sigma$ is estimated by the stream tolerance $T_1$ (see Appendix B).

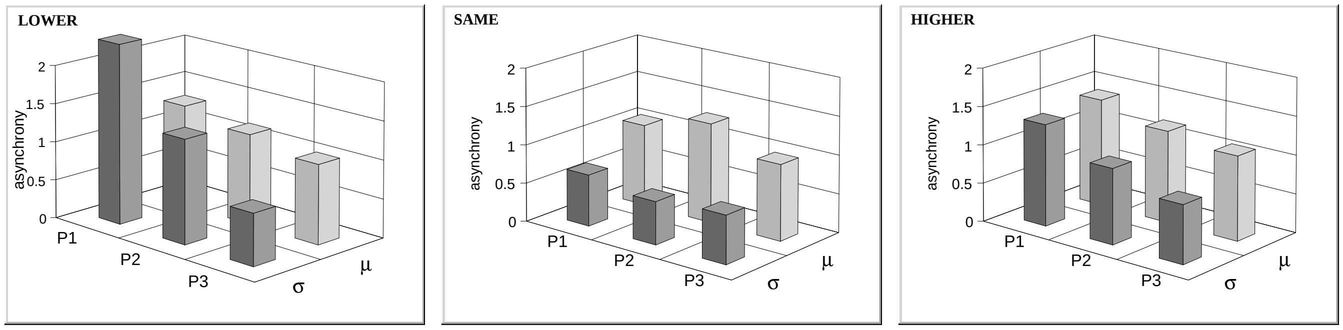

Figure 8. Comparison of mean (front row) and variance (second row) on inter-stream asynchrony for protocols P1, P2 and P3. Desirable characteristics are low mean and variance, across all load conditions. Protocol P3, based on our adaptive scheme, meets these requirements. Replay results shown for lower (left), similar (center), and higher (right) load conditions. Session object re-executed encapsulated about 250 seconds of voice-annotated, user-interactions with a MacDraw-like application.

| Protocol | $\mu_{async}$ | $\sigma_{async}$ |
|---|---|---|
| P2 (Abs Sched) | 0.49200 | 0.71007 |
| P2 (Diff Sched) | 0.41089 | 0.65735 |
| P3 (Abs Sched) | 0.59340 | 0.60323 |
| P3 (Diff Sched) | 0.54330 | 0.54015 |

Table II. Differential scheduling was found to be statistically better. Variability in asynchrony $\sigma_{async}$ is significantly reduced by the use of a differential scheduler and adaptive synchronization, as in our P3 protocol. Frequent lookups to timing services in an absolute scheduler result in greater variability than the one introduced by periodical lookups of a differential scheduler. Replay and record performed under similar load conditions.

an attempt to reset the inter-stream asynchrony to zero. The actions carried out in Step 3 are further detailed in Fig. 7. After this, a new scheduling interval is started.

Our scheduler approach has the following two benefits.

- Dependency on timing services is reduced to only synchronization events.
- Variability introduced by (frequent) timing services lookups (due to probe effects) is significantly reduced.[4]

Table II quantifies this variability component by comparing asynchrony for modified P2 and P3 protocols (using schedule departure corrections on every event) vs. both P2 and P3 (using periodical departure corrections).

In conclusion, our adaptive scheduling is based on the use periodical, statistical schedulers, running malleable logical time systems. On every scheduling interval, each stream scheduler revises its run-time schedule. Run-time schedules are adjusted to fit their corresponding stream's tolerances. This is important, because streams sharing execution on a processor may have different tolerances to asynchrony.

[4]— Our baselining experiments found that the *timing* precision of the Mach OS timing services lookups was unreliable, (i.e., lacked small variance), unless requests were at least an order of magnitude greater than Mach's base precision of $15ms$.

## COMPARATIVE ANALYSIS

The protocols described in Table I were analyzed under different background load conditions for the parameter values **given in Appendix A**, (i.e., expected scheduling interval duration $E(T) = 2.0s$). A load generator capable of creating a specifiable job-mix of CPU-, memory- and I/O-bound jobs generated the background load.

To test the ReplayKit toolkit, a MacDraw-like drawing application was made replay-aware through our toolkit. Using a baseline background load, a 256-second session with this application was recorded. Tasks included drawing figures, entering text, moving figures, grouping objects, resizing objects, scribbling, etc. Each action was voice- annotated by a remark explaining the action.

To compare the performance of the protocols, the session was replayed under several background loads: a lesser load (25% of baseline load), a similar load (100% of baseline load), and a higher load (400% of baseline load).[5] The results of this testing for protocols P1, P2, and P3 are shown in Fig. 8. Desirable characteristics are **low mean and variance** for the asynchrony, **across all load conditions**.

Under similar load conditions for record and replay, protocols P1, P2 and P3 showed similar performance (in terms of mean and variance of asynchrony), and were judged to be good enough by observers.

Under lower and higher load conditions during replay than at record time, protocol P1 exhibited higher variance in asynchrony. The audio quality continued to be good but the time interval between window events tended to drift from that at record time. Since no attempt was made at synchronization between the two streams, larger asynchrony and larger variance was observed than the other two protocols.

Comparing P2 and P3 for lower and higher loads, P3 was judged to be both qualitatively and quantitatively superior than P2. The main difference between P2 and P3 is that P3 is an adaptive protocol implementing our compensated scheduling interval strategy, (as shown in Fig. 6), on the window stream. The compensation factor $\omega$ varied between

[5]— in terms of the number of jobs waiting in the ready queue, as given by the $w$ Unix command, (baseline load index was approximately 3.5 jobs).

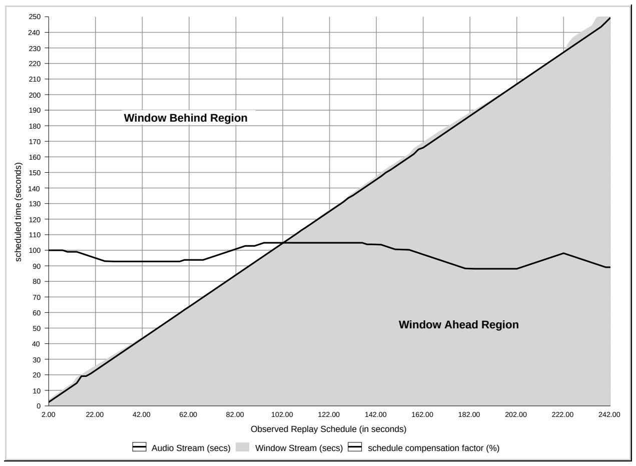

Figure 9. Detailed time-line performance of protocol P3 under higher load conditions for the same session object. Shaded area represents replay schedule for the window stream, diagonal represents replay schedule for the audio stream, and horizontal line represents compensation factor $\omega$. On each scheduling interval, the compensation factor $\omega$ adapted to trends in the measured asynchrony.

80% and 110% of the original scheduling interval.[6] These adjustments provided some compensation for timing variability during the replay of the session.

We also ran experiments with protocol P4, but we found audio playback quality to be unacceptable because of gaps introduced in the playback by the 2-WAY synchronization scheme used by P4.

Figure 9 shows the behavior of protocol P3 under higher load conditions, for one of the runs. The horizontal axis references the $i^{th}$ scheduling interval ($n \rightarrow 250$). The diagonal represents the schedule of the audio stream. The shaded area represents the compensated schedule of the window stream. A perfect diagonal match between both schedules would represent ideal scheduling and synchronization. Also shown is the compensation factor (actually, $1 - \omega$), in percentage points. An horizontal line at 100% would correspond to the use of uncompensated scheduling intervals, such as in P2. Values below 100% correspond to "speeding up" the window-event stream and values above 100% correspond to "slowing down" the window-event stream. Careful examination of the data shows that P3 attempts to slow down the window-event stream when it is continuously ahead of audio and speed it up when it is continuously behind.

### Analysis of the Results

Some very interesting conclusions, (refer to Table III), can be drawn for our protocol P3 *across all load conditions.*

- The average asynchrony $\mu_{async}$ for protocol P3 was about half the mean scheduling interval duration[7] $\mu_T$, ($\mu_{async} \rightarrow \frac{\mu_T}{2}$).

[6]— recall that a rate of 80%, for instance, would imply that in order to provide an inter-event delay of 50 ms, the timer would be set for 40 ms.

[7]— scheduling interval duration was defined by equation (2).

| Audio Frame Size | $\mu_T$ | $\mu_{async}$ | $\sigma_{async}$ |
|---|---|---|---|
| $4K \rightarrow 0.500s$ | 0.49247 | 0.22698 | 0.16709 |
| $8K \rightarrow 1.000s$ | 0.95785 | 0.52493 | 0.34628 |
| $16K \rightarrow 2.000s$ | 2.03606 | 0.96771 | 0.60582 |

Table III. Relationship between frame size and asynchrony. Mean asynchrony $\mu_{async}$ is about half the audio frame $\mu_T$. Variability in asynchrony $\sigma_{async}$ is about one third $\mu_T$.

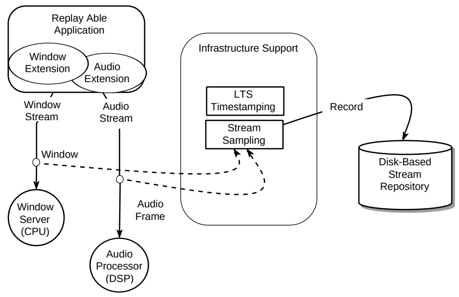

Figure 10. Application model for record of window and audio streams. The audio and window streams are sampled, timestamped and recorded into a persistent repository.

- The standard deviation of the asynchrony $\sigma_{async}$ for protocol P3 was about one third $\mu_T$, ($\sigma_{async} \rightarrow \frac{\mu_T}{3}$).

These parameters could be lowered by using smaller scheduling intervals (and thus, smaller audio frame sizes), but then the audio quality was found to deteriorate on the NeXTs because of the increased thread scheduling overheads. Finally, both parameters $\mu_{async}$ and $\sigma_{async}$ were stable across all load conditions, as one would expect for an inter-stream asynchrony random process in statistical process control.

Timing variability was modeled by random variables $f_o()$. There are two competing asynchrony random processes: the overhead functions $f_o(A)$ and $f_o(W)$, (both were discussed in Section ). For negligible $f_o()$, a simpler protocol such as P2 should work. However, for bounded, large $f_o()$, the support of synchronized replay requires adaptive scheduling support. Finally, for unbounded $f_o()$, a 2-WAY, adaptive protocol is needed.

## IMPLEMENTATION

We implemented an object-oriented prototype toolkit, REPLAYKIT, for NeXT workstations.[8] The REPLAYKIT toolkit provides the REPLAYABLE object class. The REPLAYABLE class provides applications transparent access to infrastructure services. A MACDRAW-like object oriented drawing application and a TEXT editor application were retrofitted with the toolkit. REPLAYABLE applications access toolkit features through menus and windows added to the application. The REPLAYKIT toolkit allows applications to: (1) re-execute window events (e.g., gesturing, typing, moving windows), (2) record voice-annotations, (3) provide synchronized replay of these streams, and (4) to replay selected streams. In addi-

[8]— the NeXTs run MACH OS.

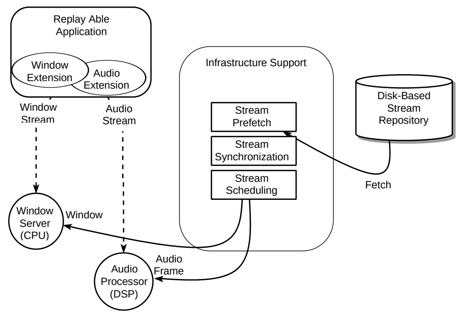

Figure 11. Application model for replay of window and audio streams. Events are prefetched and dispatched to the corresponding stream of the REPLAYABLE application. The infrastructure takes care of maintaining the relative synchronization of the streams and making the necessary adjustments to preserve relative synchronization.

tion, an API for stream capture and replay is provided.

The prototype currently supports two streams: the window and the audio stream. Each stream is dispatched to a separate processor. The window event stream is dispatched to the CPU — which is subject to arbitrary load conditions. The audio stream is dispatched to the DSP — assumed to be dedicated. These components (application, streams, DSP, CPU, infrastructure services, disk, and data paths) are shown in Figures 10 and 11. Fig. 10 shows the record-time view. Fig. 11 shows the replay-time view of the prototype. The infrastructure provides a *logical time system* (LTS) to support time-stamping of events. It also provides efficient, disk access to streams. **The experimental setup is further discussed in Appendix A.** Finally, it provides per-stream scheduling and inter-stream synchronization protocols to support faithful replay of streams.

## CONCLUSION

This paper addressed media scheduling and synchronization issues for the support of faithful re-execution of session objects when subject to timing variability. In our prototype, audio and window streams were kept synchronized during the re-execution of the session. An adaptive scheduling algorithm was designed to account for timing variability present in the replay of these streams. Smoothed forecasts and statistical process control guidelines were used to detect trends in inter-stream asynchrony.

When replay was subject to timing variability, our protocol P3 was effective in providing relative inter-media synchronization. The management of overheads $f_o(A)$ and $f_o(W)$ was achieved through integration and significant extensions to existing synchronization frameworks.

- To implement inter-media synchronization, we implemented a modified master/slave model. Synchronization was initiated by the slave streams.
- To bound inter-stream asynchrony, we used periodical synchronization.
- To remove dependencies and overheads from timing services during a scheduling interval, we used differential scheduler(s).
- To control variability within a scheduling interval, we used statistical guidelines to adapt the schedule of slave stream(s). Schedule(s) were revised on each scheduling interval and asynchrony trends were used to drive adjustments.
- To adjust slave stream schedule(s) and compensate for trends in asynchrony, we adjusted idle schedule time (in the form of inter-event delays).
- To allow adjustments to multiple slave streams, we implemented a scheduler per stream, running malleable time.
- Finally, to integrate these multiple stream schedules, we implemented relative inter-media synchronization, as opposed to absolute media timing.

Our adaptive algorithm seems well suited for use in other environments having some of the following characteristics: (a) subject to unreliable or large overhead timing services (e.g., lookups to network time services), (b) subject to dynamic, varying load conditions resulting in timing variability as seen by the media's integration processor, (e.g., varying traffic conditions, as in ATM networks), and (c) requiring application layer control of the inter-stream asynchrony.

The REPLAYKIT toolkit allows us to extend these synchronization services to other applications in general purpose workstations. To explore issues in the use of session objects, we have already started to apply the REPLAYKIT toolkit to both the UARC and the MEDRAD projects (both discussed in Section 1). Future work includes extensions to video streams and to explore the use of the paradigm in collaborative environments.

## ACKNOWLEDGEMENTS

This work has been supported in part by the University of Michigan's Rackham Merit Fellowship and by the National Science Foundation under Cooperative Agreement IRI-9216848. Our thanks for the comments provided by anonymous reviewers of this paper.

## APPENDIX A: EXPERIMENTAL SETUP

Streams execute in separate threads. The infrastructure provides two generic thread models. The sequential nature of operating systems causes parallel synchronization and relative progress to suffer. Figure 12 shows the thread model used to replay window events. During the replay of the window stream, events must be produced as well as consumed by the application itself. To simulate generation of events by the user, a producer and consumer thread pair is needed. The producer thread prefetches events from disk and puts them in the shared queue at intervals determined by the differences between event time-stamps as well as by the protocol used for inter-stream synchronization. The consumer thread gets events from the shared queue and dispatches them to the window system for replay.

Figure 13 shows the thread model used to replay audio frames. To reduce overheads in disk I/O access of audio frames, we implemented buffered sampling and prefetching of frames, (as in CMSS [30]). We baselined three important parameters that affect overheads $f_o(A1)$ during the record and replay of audio frames: (1) the audio frame size, (2)

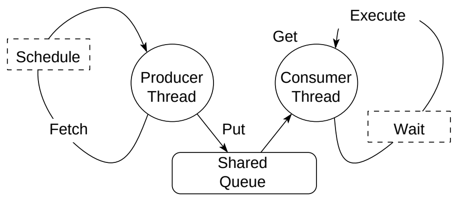

Figure 12. Thread model for the replay of the window-event stream. The producer thread fetches and dispatches events into the event queue. The consumer thread manages asynchronous re-execution of window events that were posted into the shared event queue.

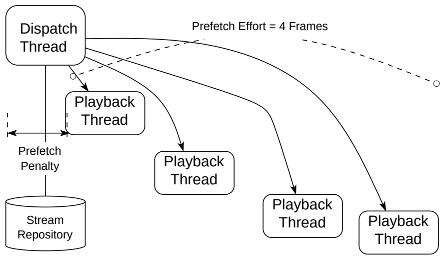

Figure 13. Thread model for replay of audio-frames. The dispatch thread amortizes prefetch overhead among several audio frames. Audio frames execute asynchronously on the audio device.

the buffering effort for writes, and (3) the buffering effort for prefetches. We summarize these baselined values below:

- Audio frame sizes of $16KB$ were found to work well on the NeXTs. Smaller frame sizes made thread scheduling overheads $f_o(A2)$ appreciable so as to affect the quality of audio playback, (see Table III). Larger values did not cause significant additional improvements in performance.
- Writing 2 frames at a time ($32KB$ of data) to the disk led to good amortization of disk overheads during recordings.
- Prefetching 4 frames at time ($64KB$ of data) was found to give good amortization of disk overheads during playback.

The above parameter values were used in the experiments described throughout this paper.

## APPENDIX B: PROTOCOL P3

The adaptive algorithm undergoes five phases, as shown in Fig. 7. The **inputs** to the algorithm are the stream tolerance to timing departures $T_1$, the past compensation factor $\omega_i$ ($\omega_0 = 1$), the stream's compensation update function $\gamma_i()$, and the past inter-stream asynchrony history between this stream and its master, $(\epsilon_0, \cdots, \epsilon_i)$. The algorithm **outputs** a new compensation factor $\omega_{i+1}$. The $\omega_{i+1}$ policy formulated by the $i^{th}$ synchronization event is applied to all window events between the $i^{th}$ and $i+1^{th}$ synchronization events.

During the processing of the $i^{th}$ synchronization event, the following actions are carried out:

- `ASYNCHRONY-MEASURE`: At the scheduling of the $i^{th}$ synchronization event on the window (slave) stream and the last known $j^{th}$ synchronization event seen on the audio (master) stream, the $i^{th}$ inter-stream asynchrony estimate $\epsilon_i$ is generated as:
  $\epsilon_i = t(audio_j) - t(window_i)$.
- `STREAM-SYNCHRONIZATION`: If $i > j$ the window stream waits until the $i^{th}$ synchronization event occurs on the audio stream. Otherwise, do nothing.
- `TREND-ANALYSIS`: This phase makes use of the stream's tolerance level $T_1$, the inter-stream asynchrony history, $(\epsilon_0, \cdots, \epsilon_i)$, to detect trends in asynchrony, under SPC $\sigma$-based guidelines. In general:
  $if\ (|\epsilon_i| > T_1)\ and\ (|\epsilon_{i-1}| > T_1)\ then$
  $if\ (isWindowAheadTrend())\ then\ \rho = \texttt{DECREASE}$
  $if\ (isWindowBehindTrend())\ then\ \rho = \texttt{INCREASE}$
  $else\ \rho = \texttt{SAME}$
- `FORECAST-UPDATE`: The replay speed for the $(i+1)^{th}$ scheduling interval is updated as:
  $if\ (\rho == \texttt{DECREASE})\ then\ \omega_{i+1} = \omega_i + \gamma_i(\epsilon_i)$
  $if\ (\rho == \texttt{INCREASE})\ then\ \omega_{i+1} = \omega_i - \gamma_i(\epsilon_i)$
  $if\ (\rho == \texttt{SAME})\ then\ \omega_{i+1} = \omega_i$
  where the stream's compensation update function $\gamma_i(\epsilon_i)$ determines an update based on the magnitude of the current inter-stream asynchrony $\epsilon_i$. We found that discrete compensation updates work better than updates proportional to the asynchrony because the latter tends to be less forgiving of random asynchrony hits. We used the following discrete step function:
  $\gamma_i(\epsilon_i) = \left\{ \begin{array}{ll} 0.001 & T_1 < |\epsilon_i| < 2T_1 \\ 0.01 & |\epsilon_i| > 2T_1 \end{array} \right\}$
- `POLICY-IMPLEMENTATION`: For every event in the window stream on the next $(i+1)^{th}$ scheduling interval, the inter-event delay time between window event $k$ and $k+1$ is set to:
  $\Delta_{k,k+1} = \{t(window_{k+1}) - t(window_k)\} * \omega_{i+1}$

## REFERENCES

[1] H.M. Abdel-Wahab, S. Guan, and J. Nievergelt. Shared workspaces for group collaboration: An experiment using Internet and Unix inter-process communication. *IEEE Communications Magazine*, pages 10–16, November 1988.

[2] D.P. Anderson and R. Kuivila. A system for music performance. *ACM Transactions on Computer Systems*, 8(1):56–82, February 1990.

[3] R. Baker, A. Downing, K. Finn, E. Rennison, D.D. Kim, and Y.H. Lim. Multimedia processing model for a distributed multimedia I/O system. In *Proc. of the 3rd Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, pages 164–175, La Jolla, CA, USA, November 1992.

[4] D.C.A. Bulterman and R. van Liere. Multimedia synchronization and UNIX. In *Proc. of the 3rd Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, pages 108–119, La Jolla, CA, USA, November 1992.

[5] D.C.A. Bulterman, G. van Rossum, and R. van Liere. A structure for transportable, dynamic multimedia documents. In *Proc. of the Summer 1991 USENIX Conference*, pages 137–154, Nashville, TN, USA., June 1991.

[6] M. Cecelia-Buchanan and P.T. Zellweger. Scheduling multimedia documents using temporal constraints. In *Proc. of the 3rd Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, pages 237–249, La Jolla, CA, USA, November 1992.

[7] G. Chung, K. Jeffay, and H. Adbel-Wahab. Accomodating latecommers in shared window systems. *IEEE Computer*, 26(1):72–74, January 1993.

[8] R. Clauer, J.D. Kelly, T.J. Rosenberg, C.E.P. Stauning, and et. al. UARC: A Prototype Upper Atmostpheric Research Collaboratory. *EOS Trans. American Geophys. Union*, 267(74), 1993.

[9] E. Craighill, M. Fong, K. Skinner, and et. al. SCOOT: An object-oriented toolkit for multimedia collaboration. In *Proc. of ACM Multimedia '94*, pages 41–49, San Francisco, CA, USA, October 1994.

[10] E. Craighill, R. Lang, M. Fong, and K. Skinner. CECED: A system for informal multimedia collaboration. In *Proc. of ACM Multimedia '93*, pages 436–446, CA, USA, August 1993.

[11] R. Dannenberg, T. Neuendorffer, J. M. Newcomer, and D. Rubine. Tactus: Toolkit-level support for synchronized interactive multimedia. In *Proc. of the 3rd Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, pages 302–313, La Jolla, CA, USA, November 1992.

[12] R.B. Dannenberg, T. Neuendorffer, J.M. Newcomer, D. Rubine, and D.B. Anderson. Tactus: toolkit-level support for synchronized interactive multimedia. *Multimedia Systems*, 1(1):77–86, 1993.

[13] A. Eleftheriadis, S. Pejhan, and D. Anastassiou. Algorithms and Performance Evaluation of the Xphone Multimedia Communication System. *Proc. of ACM Multimedia'93*, pages 311–320, August 1993.

[14] T. Fisher. Real-time scheduling support in ultrix-4.2 for multimedia communication. In *In Proc. of the 3rd Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, pages 321–327, La Jolla, CA, USA, November 1992.

[15] S. Gibbs. Composite multimedia and active objects. In *Proc. of OOPSLA'91*, pages 97–112, Phoenix, AZ, USA, October 1991.

[16] S. Gibbs, L. Dami, and D. Tsichritzis. An object-oriented framework for multimedia composition and synchronization. In *Multimedia Systems, Interaction and Applications: Proc. of the First Eurographics Workshop*, pages 101–111. Springer-Verlag, April 1991.

[17] T. Imai, K. Yamaguchi, and T. Muranaga. Hypermedia conversation recording to preserve informal artifacts in realtime collaboration. In *Proc. of ACM Multimedia '94*, pages 417–424, San Francisco, CA, USA, October 1994.

[18] K. Jeffay, D. Stone, and F. Smith. Transport and display mechanisms for multimedia conferencing across packet switched networks. *Computer Networks and ISDN Systems*, 26(10):1281–1304, July 1994.

[19] K. Jeffay, D.L. Stone, and F. Donelson. Kernel support for live digital audio and video. *Computer Communications*, 15(6):388–395, July 1992.

[20] H.P. Katseff and B.S. Robinson. Predictive prefetch in the Nemesis multimedia information service. In *Proc. of ACM Multimedia '94*, pages 201–210, San Francisco, CA, USA, October 1994.

[21] L. Li, A. Karmouch, and N.D. Georganas. Multimedia teleorchestra with independent sources: Part 2 — synchronization algorithms. *Multimedia Systems*, 1(2):154–165, 1994.

[22] T. Little and F. Ghafoor. Models for multimedia objects. *IEEE Journal of Selected Areas of Communication*, 8(3), April 1990.

[23] P. Lougher and D. Shepherd. The design of storage servers for continuous multimedia. *The Computer Journal*, 63(1):69–91, January 1993.

[24] N.R. Manohar and A. Prakash. Replay by re-execution: a paradigm for asynchronous collaboration via record and replay of interactive multimedia streams. *ACM SIGOIS Bulletin*, 15(2):32–34, December 1994.

[25] N.R. Manohar and A. Prakash. The Session Capture and Replay Paradigm for Asynchronous Collaboration. In *Proc. of European Conference on Computer Supported Cooperative Work (ECSCW)'95*, Stockholm, Sweden, September 1995.

[26] A. Mathur and A. Prakash. Protocols for integrated audio and shared windows in collaborative systems. In *Proc. of ACM Multimedia '94*, pages 381–390, San Francisco, CA, USA, October 1994.

[27] W.A. Montgomery. Techniques for packet voice synchronization. *IEEE Journal on Selected Areas In Communication*, SAC-1(6):1022–1027, December 1983.

[28] J. Nakajima, M. Yazaki, and H. Matsumoto. Multimedia/realtime extensions for the Mach operating system. In *Proc. of the Summer USENIX Conference*, pages 183–196, Nashville, TN, USA, Summer 1991. USENIX.

[29] P. Venkat Rangan and Harrick M. Vin. Designing file systems for digital video and audio. *ACM Transactions of Computer Systems*, 18(2):197–222, June 1992.

[30] L.A. Rowe and B.C. Smith. A Continuous Media Player. In *Proc. of the 3rd Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, pages 376–386, La Jolla, CA, USA, November 1992.

[31] D. Rubine, R.B. Dannenberg, and D.B. Anderson. Low-latency interaction through choice-points, buffering and cuts in Tactus. In *Proc. of the Int'l Conference on Multimedia Computing and Systems*, pages 224–233, Los Alamitos, CA, USA, May 1994.

[32] R. Steinmetz. Synchronization properties in multimedia systems. *IEEE Journal of Selected Areas of Communication*, 8(3):401–411, April 1990.