# INTERACTIVE DELAYED-SHARING OF COMPUTER-SUPPORTED WORKSPACES VIA THE STREAMING OF RE-EXECUTABLE CONTENT

by

**Nelson R. Manohar**

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1997

*"The tools and workspace of a scholar."*

Para Doña Sila Alers Ruiz, Jenny L. Goerbig, y el resto de mi querida familia,

por su paciencia y cariño.

# ACKNOWLEDGEMENTS

I would like to say thanks to God, my mamá Doña Sila Alers Ruiz, my papá Don Robert R. Manohar, my hermano Ramjeet, and the rest of my family back in Puerto Rico for everything. To Jen. To good friends, old and new, here and there, for their friendship. To my office mates, Nauman, Rob, Matthew, Amit, Hyongsop, and others. To folks at the UARC and the Medical Testbed Collaboratories. To my advisor, Prof. Atul Prakash and members of my committee, Profs. Daniel E. Atkins, Elke A. Rundensteiner, Daniel Teichroew, and Toby J. Teorey; for their time, support, and consideration — but equally important, for their kindness and fairness. To many other inspiring teachers, mentors, and professors during my years of schooling such as Profs. Kevin Compton, Yu-Hen Hu, Octavio Chavez, Ricardo Mediavilla, and Felix Alvarado. This work was supported in part by the National Science Foundation through grants IRI-92-16848 (*UARC: A Prototype Upper Atmospheric Research Collaboratory*) and ECS-94-22701 (*A Medical Collaboration Testbed*) and by the University of Michigan's Rackham Merit Fellowship.

# PREFACE

This dissertation introduces the notion of interactive delayed-sharing of a session on a computer-supported workspace to allow reuse of such session at a later time. A data artifact, referred to as a session object, encapsulates the session. A session object is composed of heterogeneous multimedia streams that represent temporally-ordered input sequences to applets in the workspace. Playback of a session object recreates the underlying workspace spanned by these applets through the streaming and re-execution of these input sequences in their respective applets. The contributions of this dissertation are as follows.

First, this dissertation pioneers the re-execution approach to the record and replay of sessions for supporting computer-supported collaborative work. In doing so, it introduces a novel paradigm for flexible support of asynchronous collaboration that allow users to collaborate by annotating, editing, and refining these delayed-shared workspaces. This dissertation explores these collaborative features particularly focusing on the record, representation, and playback of these workspaces.

Second, this dissertation introduces the notion of workspaces as authored, transportable, and temporally-aware artifacts and investigates flexible mechanisms for the delivery of temporal-awareness to workspaces composed of heterogeneous applications through the decoupling of tool and media services. This dissertation develops a formal specification through temporal relationships in these workspaces.

Finally, this dissertation describes mechanisms for the integration of re-executable content streams together with continuous multimedia streams. It describes a framework for media-independent integration of heterogeneous multimedia streams. The dissertation introduces the use of statistical process controls to guide the relaxation and/or constraint of scheduling and synchronization mechanisms so as to flexibly support a range of heterogeneous media streams and their temporal relationships.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF APPENDICES

# CHAPTER 1

# THE INTERACTIVE DELAYED-SHARING OF COMPUTER-SUPPORTED WORKSPACES

*"... there is part of ourselves in everything we do."*

— *ageless proverb.*

## 1.1 Introduction

This dissertation is about computer users, the computer workspaces they use, and the sessions they have on those. The Random House Dictionary defines a *session* as a single continuous sitting, or a period of sitting, of persons so assembled to pursue a particular activity. In the pursuit of a particular activity, sessions represent time spent in the creation of an end-goal, typically an artifact of value to others (e.g., a document, an analysis, etc.). Workspaces, on the other hand, represent the tools used in the production of such artifacts. Unlike the case of workspaces, we have lacked intuition about what it means to reuse a session between individuals. This dissertation provides intuition into such reuse of a session.

These sessions and workspaces are objects of specifiable behavior. In this dissertation, I discuss strategy, architecture, and mechanisms to support the reuse of these session objects by different users, at different times, on possibly different workstations. In particular, I present architecture and mechanisms for the capture, storage, playback, and manipulation of sessions and their underlying workspaces.

Here I describe a paradigm and its associated collaboration artifact that pioneers a new approach in computer-supported collaborative work. I refer to this paradigm as the ***Session Record and Replay Paradigm***. The paradigm introduces flexible support of asynchronous computer-supported collaborative work (herein, asynchronous collaboration) in a way such that interactive reuse of a previously recorded session is possible.

### 1.1.1 Computer-Supported Collaborative Work

Computer-supported collaborative work (CSCW) is an interdisciplinary research area devoted to exploring issues of designing computer-based systems that enhance the abilities to cooperate and integrate activities in an efficient and flexible manner for people in cooperative work situations. There are two modes of collaboration: synchronous and asynchronous. The synchronous modality refers to collaboration that occurs in real-time between two or more users whereas the asynchronous modality refers to collaboration that occurs off-line, between one or more users. For example, consider a typical meeting. Computer-supported conferencing represents a form of synchronous collaboration. On the other hand, the transcribing and e-mailing of meeting notes represents a form of asynchronous collaboration.

Synchronous collaboration, however, requires that users of these multi-user computer-supported workspaces find common time and then collaborate. In many cases, finding a common time to work between geographically distributed users is not an easy task. Consequently, a synchronous mode of collaboration can often be too imposing on the schedule of the participants. On the other hand, in asynchronous collaboration, one or more users collaborate through exchange and refinement of collaboration artifacts, without a need to find and schedule common times. Asynchronous collaboration represents a less imposing form of collaboration over the schedule of geographically distributed participants.

### 1.1.2 Sessions and Computer-Supported Workspaces

The notion of sessions and workspaces is central to synchronous computer-supported collaborative work. However, the notion has been absent from asynchronous computer-supported collaborative work. Several systems for the support of asynchronous collaboration provide ways to model the interactions among users and the evolution of collaboration repositories [41, 55, 79]. Our work represents a complimentary paradigm for asynchronous collaboration that allows users to record and replay a session. Our paradigm couples the notions of shared workspaces (found in synchronous groupware) and artifact evolution (found in asynchronous groupware) to provide an artifact that allows interaction with the evolution of a shared workspace.

Under our paradigm, a session with a computer-supported workspace (CSW) is encapsulated into a data artifact, referred to as a *session object.* The manipulation of session objects and the underlying CSW provides a new approach to asynchronous collaboration. Users collaborate by annotating, by modifying, and by a back-and-forth exchange of these session objects. The basic mechanisms allow a participant who misses a collaborative session on a workspace to catch up on the activities that occurred during the session.

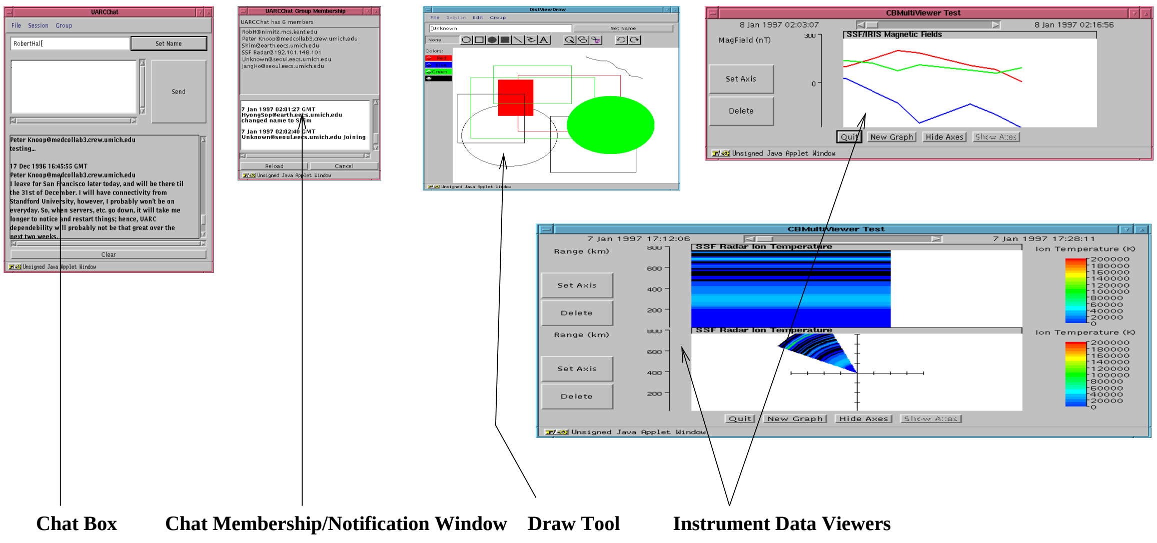
Synchronous collaboration deals with the sharing of computer-supported workspaces. Asynchronous collaboration, on the other hand, deals with the sharing of artifacts of collaboration content. One of the central contributions of this dissertation is the introduction of a collaboration artifact, the **session object**, for interactive delayed-sharing of a computer-supported workspace. This dissertation lays the ground work for supporting the features of this new artifact.

### 1.1.3 Research Motivation

At the national level, efforts are under way to deploy technology that enables researchers in geographically distributed sites to collaborate. A step in this direction has been the development of the concept of the national collaboratory [94]; described as:

> "*...a center without walls in which the nation's researchers can perform research without regard to geographical location – interacting with colleagues, accessing instrumentation, sharing data and computational resources, and accessing information from digital libraries.*"

The collaboratory infrastructure is envisioned, therefore, to support distributed interaction between people, access to remote information sources and digital libraries, and access and interaction with remote and unique facilities. The aspects of access and interaction with remote and shared facilities are the focus of the prototype **Upper Atmospheric Research Collaboratory**, or UARC [19] — a major collaboration testbed in the space science community funded since September 1992 by the *Computer Information Science and Engineering Directorate* in cooperation with the *Atmospheric Sciences Directorate* of



Peter Knoop@medcollab3.crew.umich.edu
testing...

17 Dec 1996 16:45:55 GMT
Peter Knoop@medcollab3.crew.umich.edu
I leave for San Francisco later today, and will be there til the 31st of December. I will have connectivity from Standford University, however, I probably won't be on everyday. So, when servers, etc. go down, it will take me longer to notice and restart things; hence, UARC dependebility will probably not be that great over the next two weeks.

UARCChat has 6 members
RobH@nimitz.mcs.kent.edu
Peter Knoop@medcollab3.crew.umich.edu
Shim@earth.eecs.umich.edu
SSF Radar@192.101.148.101
Unknown@seoul.eecs.umich.edu
JangHo@seoul.eecs.umich.edu

7 Jan 1997 02:01:27 GMT
HyongSop@earth.eecs.umich.edu
changed name to Shim

7 Jan 1997 02:02:40 GMT
Unknown@seoul.eecs.umich.edu Joining

**Figure 1.1: Snapshot of the interface presented by the multi-user UARC computer-supported workspace to a user. Real-time synchronous sessions with this workspace allow multiple experts to use this collaborative workspace to facilitate analysis of space science. We would like to extend the ability to browse such sessions at a later time and perhaps, continue further analysis.**

the *National Science Foundation*. The motivation for our research comes from the needs and requirements of the UARC testbed.

The UARC is a multidisciplinary joint venture of researchers in upper atmospheric and space physics, computer science, and behavioral science. UARC has developed an international networked collaboration laboratory (a "*collaboratory*") in which computing and communication technology are combined to allow geographically distributed scientists to work together. This distributed community of domain scientists has network access to a remote instrument site in Kangerlussuaq, Greenland. The UARC collaboratory operates from the United States, and is based at the University of Michigan in Ann Arbor, Michigan. Its success depends on the cooperation of many participants, including researchers from SRI International, the Danish Meteorological Institute, the University of Maryland, and the Lockheed Palo Alto Research Laboratory. Figure 1.1 shows components of the rich computer-supported workspace shared by UARC scientists.

A key observation made during early usage of the UARC collaboratory was the need to review collected data and discuss it at later times. Such a need arises because of two main reasons in collaboratories. First, because the time zone distribution of the collaborative group is typically large (e.g., California, Demmark, Greenland, Michigan, and Maryland), finding common time between scientists is difficult. In this case, if a participant missed a collaborative session, the ability to replay such session would make possible for the participant to review the collaborative content at a later time. Second, because of its nature, a collaboratory involves investigators from many academic disciplines (for example, space scientists, process observers, computer scientists – and their students). These participants have typically different, and possibly conflicting, research goals with respect to a given collaborative session. Here, the ability to review a session at a later time might prove essential so as to allow them to complete their different research goals at a later time (for example, by supporting (respectively) *off-line* discussions about a session by space scientist, analysis of the dynamics of a session by observers, baselining functionality used in a session by computer scientists, and/or assimilation of valuable experience captured within a session by students). Thus, tools that support synchronous (real-time) interactions need to be designed to support asynchronous collaboration. Such tools could, for example, permit a researcher to playback data from various instruments collected during a previous data collection interval, annotate the data, discuss the results using voice, and encapsulate all of this into a file that could be sent to a colleague who could playback the discussion on his or her own workstation. The colleague could edit the file to add value by including comments and annotation, and send the file back. 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.

### 1.1.4 Approach to Session Record and Replay

Fig. 1.2 shows a high level view of the capture and replay of an interactive session with an application. During the capture of the session, inputs and user interactions with the application, audio annotations, and resource references (e.g., fonts, files, environment) are recorded into a *session object* (Fig. 1.2a). During replay, these inputs are re-executed. The replay of the session uses data stored in the session object to simulate the original session (Fig. 1.2b).

Session objects are re-executable artifacts. The replay of session objects re-executes inputs over the state of applications in a workspace. This approach exhibits benefits that

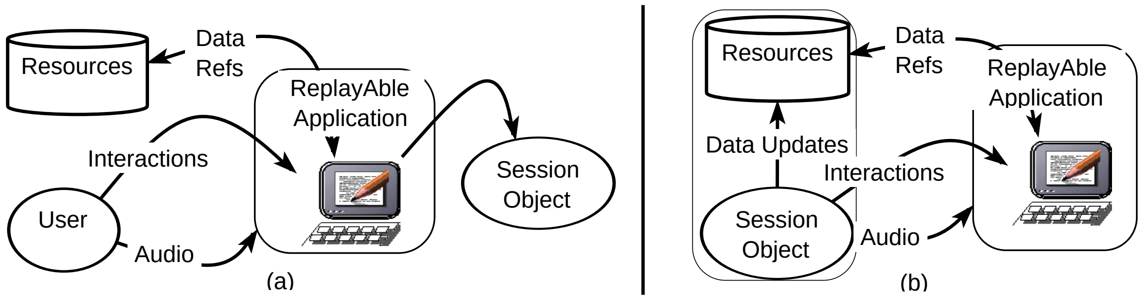


**Figure 1.2: Capture and replay of an interactive session with a Replayable application. During the capture of the session, user interactions, audio-based annotations, and resource references are encapsulated into a persistent, transportable, session object. During the replay, the session object makes these accessible to the application.**

are of particular interest in CSCW:

- interactivity over intra-task content:

  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 on the asynchronously-shared CSW; for example, interactions *between* sessions and even *within* a session are possible.

- high content fidelity:

  Because the original CSW is recreated and re-executed (rather than re-displayed as in a digital movie), the highest possible fidelity of replay is achieved, which in some domains is essential — for example, consider our medical collaboration testbed, described below.

- reduced artifact size:

  Because inputs (rather than outputs) are captured, session objects have relatively small size (this is argued on Chapter 3). Small size reduces the communication overheads associated with the exchange of collaboration artifacts between remote collaborators and can facilitate the exchange of artifacts between collaborators.

**Discussion**

The ability to interact with the *re-creation* of an artifact is important whenever the process of reaching a conclusion contains information that facilitates the understanding of the conclusion itself. One such example is the process of diagnosing an x-ray by a radiologist. This process creates a collaboration artifact, the diagnosis, shared by radiologists and physicians. However, the process of reaching the diagnosis also contains information of interest to physicians and other radiologists. If a session is to be re-created and interacted with, it is important that its contents represent a faithful recreation of the original session. Finally, if session objects are to be exchanged among collaborators, large artifact size reduces the portability of such collaboration artifacts. Our session record and replay approach provides interactivity over intra-task content, faithful recreation of content, and small artifact size.

## 1.2 Goals and Requirements

The following are the goals considered in delivering the paradigm:

- faithfulness:

  Regardless of playback conditions and playback platforms, the playback of a session object must be faithful to the original recording of the session.

- interactivity:

  Users must be able to successfully manipulate the artifacts in ways that capture and lead to collaboration. As has been the experience with the use of e-mail for collaboration among a group, we expect that the users will need features such as the ability to exchange, edit, annotate, and interact with session recordings. Some of the features above are similar to those found in the VCR metaphor.

- collaboration content:

  Different collaborators may be interested in different segments of a session. It is desirable for users to be able to statically browse session contents for events of interest. For instance, a user-interface analyst might be interested in browsing a recording to determine when a particular command was issued or whether a selection was made or *even considered.*

**Discussion**

The use of re-executable content for the representation of session objects impacts faithfulness, interactivity, and collaboration content requirements. First, meeting our faithfulness requirement requires accounting for causality and asynchronous event playback during the streaming of re-executable content. Second, interactive operations such as editing, annotation, and fast forward of a session object require mechanisms that account for the asynchronous and stateful nature of re-executable content. Last, the analysis of collaboration content requires mechanisms for abstracting and parsing re-executable content streams.

## 1.3 Assumptions

The re-execution approach relies on assumptions about issues on (1) the determinism of applications being replayed, (2) the existence of mechanisms for the capture of application state, and (3) the handling of resource references between record and replay workstations.

### 1.3.1 Determinism

Determinism is central to any re-execution approach, as inputs are to be applied over the state of an application. A deterministic application must respond to such stimuli in the same way, all the time. The following assumptions characterize such class of interesting deterministic applications:

- First, the inputs (i.e., state and events) that affect the computation can be captured.
- Second, the same application is available during record and replay of the inputs.
- Third, the application performs only deterministic computations over these inputs.
- Last, external resource references used by the computation can be treated as stateless references by the computation.

Versioning is another issue that relates to determinism. The re-execution approach is not only application-dependent; it is also version-dependent. If the underlying application is modified, replay of an stored session object may no longer be faithful to the original session. One solution to dependencies on the version of the application is to define the

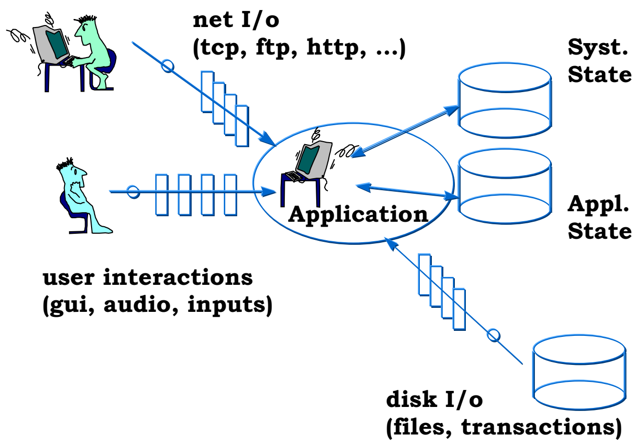


**Figure 1.3: The general application model for workstation-aware applications. Inputs and interactions are very diverse; state checkpointing is hard; and references can be made to arbitrary resources.**

session object $S$ as composed of both data (i.e., the data streams $\{S_1, \cdots, S_n\}$), and the replayer (i.e., the *replayable* application $R_{app}$). Equivalently, $S = (R_{app}, \{S_1, \cdots, S_n\})$.

### 1.3.2 Statefulness

The replay of session objects is accomplished through re-execution. Re-execution of events may be causally dependent on the state of the application prior to the start of the capture of the session. The state of the application is composed of the collective state of the application and the resources that interact with it (see Fig. 1.3). In our most basic model, a user interacts with a single application. In more elaborate models (presented in the later chapters), a user interacts with a workspace containing multiple applications.

Applications may be local or distributed across workstations; they may have open or closed interfaces to state checkpointing; etc.. In general, *state checkpointing at arbitrary points* of arbitrary applications in a computation is a technically hard problem in workstation-central applications. When state capture at arbitrary points is not possible, statefulness is removed by recording and replaying from the very start of the applet(s) and forward on. In this case, session objects can be regarded as an interactive log of the provisioning of one or more service(s) to a user.

There are several approaches to the *initial state capture problem.* One approach is to assume that capture of application state is application-dependent and an application responsibility as for example, in GroupKit [88]. Another approach is to capture all inputs from the very start of the application launch. This way, the need for initial state capture is removed. The capture of state is related to the *late join problem.* The late join problem has been addressed in the implementation of the XTV system [17] by removing the need for state capture by recording all events processed from start-up and on. However, this solution results in sessions for which its size is now a function of both the duration of the session as well as of all the activities that occurred *before* the start of the session. The size of such session objects can be large, so strategies to reduce its size are needed. In XTV, this problem was addressed by attempting to identify and remove redundant or non-stateful events from the event stream.

### 1.3.3 Resource Transparency

The re-execution approach also introduces problems with the handling of resource references (e.g., fonts, files, devices, etc.) during the recording of a session. On the replay platform, referenced resources may be unavailable and, even worse, if available, may not be in the same state.

To address the unavailability problem, resource references are classified as being *public* or *private.* Public resources are assumed to be widely available across platforms. Private resources need to be made available to replay platforms. This is known as the *resource mapping problem.* This problem has partly been addressed in X pseudo servers, such as Xtv [1], through the use of canonical mapping schemes. The canonical mapping scheme maps resource references made by a sharing-provider display (for example, window identification numbers, font sizes, etc.) into canonical resource references that are de-referenced into local resource references by sharing-subscriber displays.

To address the *stateful resource problem*, resource references can also be classified as being *stateful* or *stateless.* Stateless resources should be made available only when classified as private resources. Stateful resources must be available under a consistent state to replay platforms. When an initial reference to a stateful resource is made, the stateful resource must be made available to the session object (for example, by simulating access to a copy of the resource).

### 1.3.4 Discussion

Are these reasonable assumptions? The answer is yes and no. Assumptions such as state capture and resource transparency are likely to be infeasible in non-object oriented workstation-aware applications (see Fig. 1.3). However, the widespread use of object-oriented languages has made the problem of state capture for object hierarchies more accessible. Though these assumptions disqualify a number of large, monolithic, workstation-aware applications; they can be met for a type of small applications known as downloadable applets found in the internet domain.

In next generations of internet services, based on the notion of the streaming of downloadable content (for example, see Fig. 1.4), state capture, process mobility, and service/platform transparency represent fundamental building blocks of next generation architectures as found on visions of the future of the internet [11, 54, 56]. One such example is found in Birnbaum's *pervasive computing*, client-utility services [10, 11], a vision of the next generation of internet on which the state of the services resides at the service providers far on the internet cloud. In this model state capture, platform independence, and process/context migration, go hand in hand, as client terminals are re-execution devices for network state and inputs. The streaming of re-executable content is also found in next generation internet downloadable content technologies such as with the program capsules of Tennenhouse's proposed *active networks* [102] on in Kleinrock's notions of *nomadicity* [53]. In general, the streaming of re-executable content arises whenever a remote repository is used to store state and input behavior of one or more distributed computer appliances, for example in internet computers, in the next generation wireless (mobile/nomadic) computing appliances. As the quality of these digital consumer appliances increases, the need for policies to integrate richer and more complex forms of interactions (i.e., re-executable content) will become evident.

The playback of re-executable content streams, together with continuous streams, arises during the streaming of inputs to multiple applets in a workspace. This problem is also of strong interest to next generation multimedia frameworks as found for ubiquitous computing [104], nomadic computing [53], and wireless ATMs [36], with the introduction of time-dependent streaming of re-executable content. The media integration and streaming mechanisms on these frameworks face media heterogeneity. Streaming and integration re-

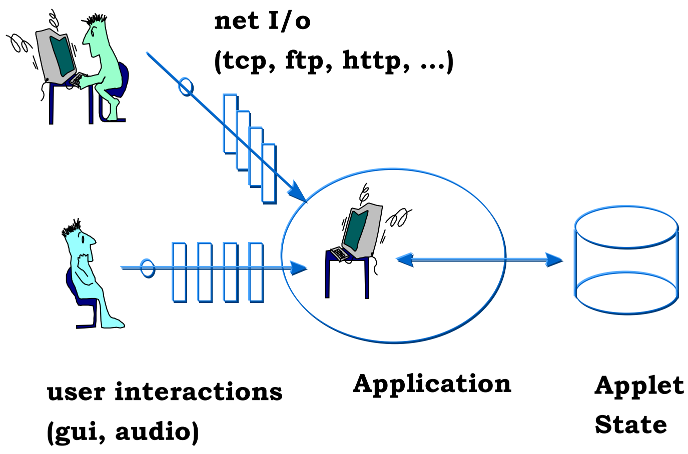


**Figure 1.4: The simplified applet model for workstation-unaware applications. Inputs and interactions are precise; state checkpointing is simpler; and external resource references are well-specified (e.g., universal resource locators, etc.).**

quire us to characterize stream requirements. For continuous media streams we possess a strong characterization of their requirements; on the other hand, for re-executable content streams, we lack a suitable characterization that addresses <u>both</u> synchronization and continuity. This is a primarely due to the burstiness, statefulness, and unpredictability of these new and (yet to be defined) re-executable content streams.

## 1.4 Overview of Research Issues

Here I compare the research horizon of issues related to the record and replay of computer-supported workspaces.

### 1.4.1 Authoring of Sessions

Session objects are complex, authored artifacts. Session objects are composed of finely-grained, time-dependent, heterogeneous streams (for example, window and audio streams). Such artifact content has substantially different characteristics from existing artifacts. For example, in a session object composed of window and audio streams, the window stream represents a stateful, aperiodical, discrete, and asynchronous stream. On the other hand, its audio stream represents a stateless, periodical, continuous, and synchronous stream. Because of the media richness being supported, even common features found in other time-dependent artifacts are substantially more complex.

As with any artifact, session objects ought to be browsed and interacted with, too. Browsing of session objects includes operations such as fast forwarding and fast replaying through a session. Browsing of session objects also comprises operations such as non-linear access (e.g., go-to operations) and static content analysis and querying. The use of re-executable content streams impacts the realization of mechanisms for collaboration. Beyond record and replay, the implications of re-executable content over other collaborative features such as browsing controls are varied. For example, the simple fast-replay requires schedule transformations that account for asynchronous execution, the more complex fast-forward requires a state jumping mechanism, and static browsing introduces an open research problem in the parsing and understanding of re-executable content. Research is needed to design artifact manipulation features that account for the richly heterogeneous media contained in session objects. Extending and evaluating these features to an application base that is available to a cooperating user community should provide insight and feedback about our design and paradigm.

### 1.4.2 Integration of Heterogeneous Media

Session objects are transportable multimedia objects. The re-execution of events is asynchronous (due to performance differences in playback on heterogeneous workstations, varying load conditions, timer inaccuracies, etc.). During replay, the playback of asynchronous streams (such as the applet inputs, window events, etc.) must be kept synchronized to the playback of continuous streams (for example, the audio stream). That is, synchronization must be performed *wrt* the relative progress of a stream and not *wrt* the progress of schedule time.

The support for heterogeneous streams and in particular the integration of re-executable content requires researching its impact over the following issues on modeling and representation:

- the design of a media-independent representation and features for the manipulation of session objects.

| Definition | *audio frame* | *voice phrase* | *applet datum* | *applet operation* |
|---|---|---|---|---|
| Continuity | Y | N | N | N |
| Statefulness | N | N | Y | Y |
| Periodicity | Y | N | Y/N | N |
| Time-Variant | N | N | Y/N | Y |

**Table 1.1: Playback of audio and re-execution of applet inputs introduces heterogeneity into the synchronization problem. These dissimilar characteristics affect our choices for synchronization.**

- the design of media integration mechanisms (e.g., scheduling and synchronization) for richly heterogeneous streams.

### 1.4.3 The Streaming of Re-executable Content

Because of the heterogeneity of the media contents of our session objects, adaptive strategies for integrated playback of continuous media streams (i.e., composed of stateless, periodical, and synchronous events) do not efficiently address the integration requirements of re-executable content streams (i.e., composed of stateful and asynchronous events). For example, mechanisms such as dropping/duplicating events have very limited applicability, during the handling of re-executable content streams. Re-executable content streams are significantly different from continuous media streams. They are more susceptible to variations on load and platform performance and their events are stateful and asynchronous. These heterogeneous characteristics strongly affect the formulation of a feasible stream synchronization solution. The support of re-executable content of arbitrary workspaces requires researching mechanisms that address causal relationships between streams and the enforcement of $n$-ary relationships between richly heterogeneous streams be supported.

### 1.4.4 The Notion of Applications as Replayable

Should applications be inherently replayable? We need investigate whether the notion of the application as a replayable object represents a new object pattern [91] that is yet to be found in other domains. An efficient way to deliver this replayable object pattern to an application is needed. One possible solution centers around object orientation. An object-oriented replayable class and its compliance protocols should allow an application to become *replay-aware*. Delivery of the replay-awareness functionality occurs then through subclassing, which extends transparent access to the replay-awareness middleware. Our is goal here is to deliver replay-awareness functionality while reducing reliance on operating system and hardware. Preliminary prototypes show that, for session objects composed of window and audio streams, this is possible. Pursuing this approach for other richly heterogeneous streams imposes new research challenges of general interest.

### 1.4.5 Replayable Workspaces

The approach allows an interactive session with a replayable application to be captured and replayed. In general, the capture and replay of sessions on which interactions span multiple applications pose new challenges. One approach to the integration of heterogeneous applications is centered around extending my temporal-awareness to an intermediary representation or perhaps a *software/message bus* such as HP's Softbench [22]. Such approach requires addressing the following issues:

- Intra-tool session capture and replay. First, the tools interfacing to the software bus must be made *replay-aware* (i.e., capable of capturing and replaying tool's sessions).
- Inter-tool session capture and replay. Second, the software-bus itself must be made *replay-aware* (i.e., capable of capturing and replaying the interactions between the tools).

Clearly, the generalization of the mechanims to record and replay to workspaces composed of multiple applets is of interest to us.

### 1.4.6 Management of Collections of Session Objects

Because session objects attempt to be *faithful* to the original session, session objects may be regarded as *active documentation* of the activities that occurred during the recorded session. Session objects are digital artifacts. A collection or library of such active artifacts has reuse and knowledge-capture value to organizations. This can be regarded as a *digital library* of process descriptions — i.e., the active artifacts. Ways to efficiently browse and query large collections of these active artifacts are needed. Requiring the replay of a session just to determine its contents is a strategy that does not scale up to larger number of sessions. The use of textual descriptions or abstracts of a session have the inherent problem that there is no intrinsic relationship between the description and the content of the session. To address these difficulties, a *markup language* for time-dependent data, such as HyTime [80], can be used.

The use of a markup language allows standardized keyword and content querying of a stream contents. For some streams, in particular the window stream, its contents may be easier to retrieve and classify, than for other streams such as the audio stream. For other streams, such as digital video, content retrieval techniques are being researched (as in [3, 37]). However, because the intended domain for markup languages is passive artifacts, as opposed to active artifacts such as session objects, some extensions might be needed. For example, consider the resource mapping problem. Through extensions to catalog, retrieve, and modify resource references found inside a reusable session object (perhaps stored in a collection) it might be possible to transform a *generic process description* into a locally executable process instance (i.e., the re-executed session object).

## 1.5 Focus of this Dissertation

The focus of this dissertation is not on the capture of application state, nor the versioning and resource dependency problem, nor the manipulation of collections of session objects, nor to further explore the usability of the paradigm and/or session objects. Existing solutions or approaches to these problems, as discussed before, can be integrated into the solutions presented herein. The manipulation and querying of collections of session objects, is however, an open research issue.

The focus of this dissertation is to explore and enrich the support for interactive delayed sharing of a computer-supported workspace. In particular, I introduce the notion of replay-awareness. In doing so, I address its requirements over the support of heterogeneous media streams, the record and replay of multiple tools, and the delivery of media and tool transparent replay-awareness to a workspace spanned by multiple tools. The result is a framework for session objects that addresses the record and replay of sessions, the integration of heterogeneous media, the streaming of re-executable content, and the notion of replayable applications and workspaces.

## 1.6 Contributions

This dissertation proposes a novel approach to computer-supported collaborative work. It introduces interactive delayed-sharing of computer-supported workspaces. The goal is to capture and replay a session and to provide interactivity over the session at a later-time. During capture, the state of, and inputs to, applets in a workspace are recorded. During playback, the workspace spanned by these applets is re-created by scheduling and re-executing state and inputs to applets as temporal media streams. Furthermore, to enhance the collaborative content of the playback, audio-based comments during the capture of the session are also recorded. The playback and manipulation of a workspace creates delayed-sharing (an interactive, asynchronously-shared workspace). Interactive delayed-sharing represents a novel approach to asynchronous collaborative work that is very different and yet complementary to traditional workflow systems. This dissertation introduces the notion of computer-supported workspaces as artifacts of collaboration that can be authored, exchanged, reused, and refined at a later time for collaborative purposes.

The playback of these re-executable content session objects requires the streaming of applet inputs and states. An applet input stream is composed of time-stamped applet events such as state snapshots and inputs. The playback of an applet input stream is accomplished through the re-execution on an applet. Synchronized re-execution of multiple applets recreates a session on the workspace spanned by the applets. Through re-execution of applet inputs enable interactivity over the playback of a session and the computer-supported workspace it spans.

The basic re-execution problem can be reduced to a temporal ordering problem. During capture of a session, inputs and state to each applet are time-stamped and stored as applet input media streams. Since applets are not distributed across workstations, applets and their streams share a common logical time system. Thus, during capture of a session, a full temporal order exists between events across different applet input media streams. Because re-execution of applet input events — herein, applet events or applet inputs. is asynchronous and likely to occur on different machines (or even when in the same machine, under variable load conditions), the playout time of applet events is unpredictable. Consequently, during playback of a session, only a partial temporal order exists between applet events across multiple applet streams. We require synchronization protocols that ensure that a full temporal order guarantee exists during playback.

Although the playback problem can be formulated as enforcing a full temporal order between partially ordered streams, the satisfaction of this goal is constrained by the individual requirements of the applet input streams, such as sensitivity to jitter or variations over the continuity of the playback of a stream. We normally refer to these as continuity and synchronization. This dissertation contains mechanisms to enforce temporal order of asynchronous events with variable tradeoffs of synchronization and continuity for fine event granularity. This dissertation builds on those media-independent mechanisms to extend replay-awareness to a workspace spanned by multiple applets.

A re-executable representation introduces several problems. The streaming of applet input represents a new kind of temporal stream. I refer to them as a re-executable content stream. Re-executable content streams are significantly different from continuous media streams. They are more susceptible to variations in load and platform performance. Their events are also stateful and execute asynchronously. These heterogeneous characteristics affect the formulation of a feasible synchronization solution. The use of re-executable content streams also impacts the realization of mechanisms for collaboration. Beyond record and replay, the implications of re-executable content streams over other collaborative features such as browsing controls are less obvious. For example, the simple fast-replay requires accounting for asynchronous event execution; the more complex fast-forward requires the placement and use of state snapshots; editing of a session requires a branching mechanism that accounts for statefulness; and static browsing introduces an open research problem in the parsing and understanding of re-executable content. My research makes enabling contributions to the above problems. These claims are revisited on Chapters 3, 5, 6, and 7.

Differences in processor performance introduce asynchrony trends during the playback of re-executable content streams. To remove these long-term playback asynchrony trends, I introduce a scheduling mechanism driven by statistical process controls that control the magnitude of compensations over the inter-event delay time that exist between asynchronous events in a re-executable content stream. Furthermore, to support the streaming of heterogeneous media, I propose a media-independent architecture for providing flexible tradeoffs between synchronization and continuity. Tradeoffs are achieved by varying the tolerance region of the scheduling and synchronization controls through statistical process control guidelines. I describe statistical quality controls to provide varying degrees of control over the quality indicators of continuity and asynchrony. These claims are examined in Chapters 4, 8, and 9.

My research provides enabling contributions to the areas of collaborative systems as well as on multimedia systems. I envision my research reaching maturation when applied to domain-specific problems such as in interactive training systems, audio-annotated interactive lesson workbooks, and transportable interactive demos. This vision is validated by the technology transfer of my research, for the computer radiology domain, at the Medical Collaboration Testbed at the University of Michigan. This vision is also applicable to the research and development of internet-multimedia application frameworks for the support of this and other collaboration forms in next generation internet services such as for interactive asynchronous distance learning and mobile workspace computing.

## 1.7 Concluding Remarks

The web, the graphical intensive component of the Internet, is strongly asynchronous. By asynchronous, I mean that interactions between users are delayed in time and collaboration is based on the exchange of artifacts (such as movie clips and images). The richness of these artifacts (i.e., their media content) determines the expressive power of the web to support asynchronous collaboration. In my dissertation, I introduce enabling technologies for a new kind of artifact for the support of asynchronous collaborative work fitted for web-based delivery.

My work allows the capture and replay of a session on a workspace in such a way that interactivity and reuse over a session at a later-time is possible. Such later-time interactive manipulation of the underlying workspace represents a new form of asynchronous collaboration, the interactive delayed-sharing of a workspace artifact. I designed mechanisms, abstractions, and frameworks for the record and replay of workspaces and their sessions at a later time. In the process of supporting such replayable workspaces, I exposed a number of novel research ideas that position the notion of replayable workspaces as an open and strong area of research. These are outlined below.

The rest of this dissertation is structured as follows. In Chapter 2, I discuss related work to my research. In particular, I address the relationship of my work to two well-established areas: multimedia systems and groupware systems. In Chapter 3, I present the paradigm for the capture and replay of sessions and its support for asynchronous collaboration. Here, I explore, define, and analyze key issues on the support of the asynchronous sharing of the workspace of a single application. In doing so, I introduce a novel data model to support the integration, streaming, and VCR-like manipulation of heterogeneous media — herein, the reference to heterogeneous media/streams refers to the integration of multiple continuous media and re-executable content streams. In Chapter 4, I describe application-level multimedia mechanisms to support the record and replay of two heterogeneous streams: a re-executable content stream and a continuous media stream. In doing so, I introduce a novel approach to multimedia synchronization that relies on a statistical process control approach to media integration.

In Chapter 5, I introduce the notion that, not only applications, but also the workspaces spanned by multiple applications should be considered replayable objects, too. In Chapter 6, I describe a flexible architecture to support replay-awareness services that are unaware (i.e., transparent) of the applets that compose a workspace. In Chapter 7, I examine media-independent specification and modeling of replayable workspaces so as to facilitate the integration of heterogeneous streams on a replayable workspace. In Chapter 8, I describe the mechanisms that support media-independent integration of multiple heterogeneous streams and their relationships on a replayable workspace. In Chapter 9, I re-examine key issues on my experience on developing replayable applications and designing replayable workspaces. Finally, in Chapter 10, I present my concluding remarks and identify exciting future directions in this emerging area of research.

# CHAPTER 2

# RELATED WORK

*"... those who do not learn the past are bound to repeat it."*
*— on the Napoleonic and Germanic invasions to Prussia.*

## 2.1 Introduction

The contributions of this dissertation relate to two main areas of research: computer-supported collaborative work and multimedia systems. Collaborative multimedia is centered around the use of rich content to support richers forms of collaboration.

With respect to collaborative systems, this research relates to shared workspaces and artifact-based collaboration systems. With respect to multimedia systems, this research relates to media integration, (e.g., modeling, specification, scheduling and synchronization) of heterogeneous temporal media streams. Below, I analyze related work on these areas.

## 2.2 Computer-Supported Collaborative Work

There are two modes of collaboration: synchronous and asynchronous. The synchronous modality refers to systems for which collaboration occurs in real-time between two or more users, the asynchronous modality refers to systems for which collaboration occurs off-line, between one or more users. Synchronous collaboration requires that users first find common time and then collaborate. A synchronous mode of collaboration can often be too imposing on the schedule of its participants. In many cases, finding a common time to work between geographically distributed users is not an easy task. On the other hand, in asynchronous collaboration, one or more users collaborate through exchange and refinement of collaboration artifacts, without need to find and schedule common times. Asynchronous collaboration represents a less imposing form of collaboration over the schedule of geographically distributed participants.

The notion of session and workspaces is central to synchronous computer-supported collaborative work [92]. However, the notion has been absent so far from asynchronous computer-supported collaborative work. Our replayable workspace paradigm couples the notions of a shared workspace (found in synchronous groupware, e.g., XTV [1, 17] TkReplay [25], DistView [83], SharedX [38], XMX/XTRAP [49], etc.) and artifact-based collaboration (found in asynchronous groupware, e.g., Conversation Builder [50], Strudel [95], gIbis [20, 21], Quilt [33], Prep [78, 79], Active Mail [41], Object Lens [55, 64], etc.). Replayable workspaces provide an artifact that allows asynchronous sharing over the refinement and evolution of a shared workspace.

The paradigm of session capture and replay for asynchronous collaboration is centered around the exchange and refinement of a work-in-progress, an artifact — (hence the names artifact-based collaboration and workflows). Many systems support asynchronous collaboration [20, 21, 41, 50, 55, 64, 95]. Such groupware systems work first, by defining a shared object (i.e., the artifact) and second, by formalizing a protocol (i.e., the workflow specification) that formally specifies transitions and states of the shared object. For example, consider gIbis and Conversation Builder, its shared object is an argumentative tree to which a well-defined set of transitions and states (the states of a multi-party design dialogue) are associated.

Our paradigm introduces a new data artifact to artifact-based collaboration (workflow) systems []. The artifact design is complementary to such systems — it can be used to incorporate intra-task content in workflow systems. Within the context of asynchronous collaboration, intra-task content (as opposed to inter-task content) refers to the process of how a decision was reached. Inter-task content, on the other hand, refers to the decisions themselves.

Annotating and editing provide a fundamental mechanism of collaboration between asynchronous collaborators: versioning of a shared artifact through iterative exchange and refinement. Delayed-time collaborative groupware systems (e.g., Siemens's [43], GMD's

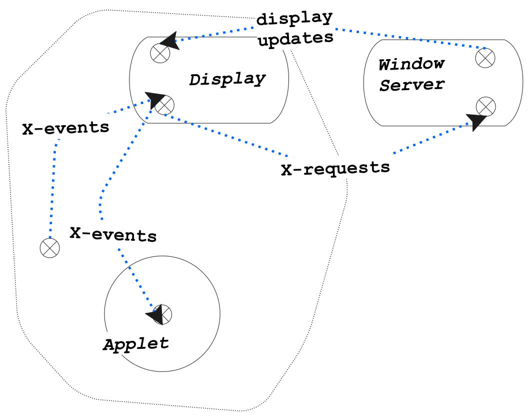


**Figure 2.1: Shared Windows and Window Servers. The X-windows processing environment (i.e., X-clients, X-client Display, and X-Window Server) deals with three kinds of inter-related events: X-events, X-requests, and display updates. Shared window servers use events from one of these streams to allow the sharing of the user interface of one or more X-clients.**

Dolphin [100], Lotus's Notes, etc.) allow users to work asynchronously via annotations added to a shared object (a document composed of rich media parts). My approach builds on those ideas but, in our case, the annotations are over an active content artifact, a recording of interactions with a workspace. Editing and annotation now are performed over an active content artifact that represents an active log, a re-executable process description.

## 2.3 Delayed-Sharing of Workspaces

Shared window systems such as XTV [1, 17] XMX [6], Shared-X [38], and window-stream oriented systems for record and replay such as CECED [24], XTRAP [49], TkReplay [25], SCOOT [23], as well as display-based screen camcorders such as Lotus Screen-Cam, and MS Camcorder, in principle, allow replay of unmodified applications. These X-based systems work, work coupled with the Window Server on an workstation in one of two ways (see Fig. 2.1). Under the first approach, (followed by systems such as XTV), the Shared Workspace Systems (SWS) sits between the applications and Window Server intercepting input events (either X events or X-requests) sent to the Window Server. The SWS uses this stream of events to reconstruct the workspace on other workstations by forwarding and re-mapping this input stream to other shared Window Server. The second approach intercepts

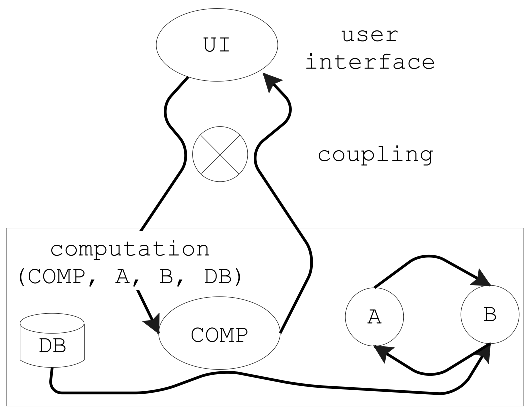


**Figure 2.2: The notions of sharing the interface of an application (UI) from sharing its computations (COMP) are very different, in particular when dealing with complex computations and delayed-sharing. In general, a complex computation is associated to resources and data (A, B, DB). Delayed-sharing requires preserving the statefulness on COMP on a different workspace, at a different time.**

output events (display updates) from Window Server to applications being shared. These systems, in principle, allow events to be replayed for specific applications.

Though replay of specific and unmodified applications is very desirable, it comes at a cost. To understand this, we need to separate the notions of sharing the interface of an application (i.e., the look and feel of a program) from sharing its computations (the underlying abstract base machine that supports the program). This difference, in the context of synchronous sharing of workspaces, was first examined by Dewan [27, 28, 29] (see Fig. 2.2). In general, a complex computation can be associated to resources (say A and B) and data (say DB). Delayed-sharing requires preserving the statefulness on COMP on a different workspace, at a different time. In the aforementioned SWSs, the object of sharing is the user interface (UI) of the application and not the application's workspace. Interactions with the underlying computational workspace (COMP) in these SWSs is thus limited. If the underlying workspace is to be shared, it is necessary to have access to the application's computation. In my approach, this access is found in the interfaces to the applet's abstract base machine. The notion of abstract base machines used here corresponds to Parnas [82].

However, this approach has other limitations. First, capturing the state of Window

Server can be difficult. Second, replay is limited to only those events that go through Window Server. The lack of state capture coupled with the lack of media integration mechanisms (so as to handle other media streams such as audio) discourages the use of existing SWSs *wrt* the requirements for the interactive delayed sharing of a workspace.

Application-dependent record and replay of a workspace has recently been proposed in the CECED system [24] through its SCOOT project [23]. The SCOOT project proposes two approaches to replay of a session. The first approach centers around periodical recording of application state snapshots. The second approach logs invocations to object's methods. This is very similar to our replay by re-execution approach. However, the SCOOT proposal does not address the synchronization of other media streams such as audio *wrt* the streaming of such re-executable content.

## 2.4 Authoring of Multimedia Sessions

Our research has similarities to multimedia authoring [14, 15] (e.g., stored media integration, interactive browsing); however, the problems are significantly different.

Authoring systems, such as MacroMind Director, allow specification of interactions over stored digital media such as video, audio, text, and animation. In media authoring, synchronization relationships are explicit (i.e., manually *crafted* via scripts or flowcharts so as to balance media processing of data intensive segments, such as movie clips, to less demanding ones, such as text-based interactions). In our artifact generation (i.e., the capture of interactive CSW sessions), synchronization relationships are implicit (i.e., a user-transparent process). Although our research then seems similar to interactive multimedia presentation research, I focus on fine-grain integration of heterogeneous media streams as opposed to orchestration of heterogeneous document parts.

Issues in specification and presentation of multimedia documents have been addressed in Firefly system [15], CMIF [14], and Isis's elastic time [52]. 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 their focus. In the work found on this dissertation, the focus has been placed on enforcing fine-grain synchronization at internal points during the streaming of a new media part (i.e., the session object). The mechanisms found in this dissertation can enhance authoring frameworks that are interested in supporting the streaming of re-executable content media parts.

Replay of heterogeneous media is addressed in Isis [52] and Firefly [15] through online (i.e., run-time) optimization. Their run-time scheduling uses linear programming to merge a hard schedule (for time-invariant streams) with relative, auxiliary schedules (for each asynchronous streams). Their run-time schedule construction is input-size dependent. Thus, implicitly, this linear program relies on the coarseness of media parts. As the number of events in a stream increases, run-time schedule construction overheads become more expensive. On the other hand, our online (i.e., performed during run-time) schedule construction is statistical and independent of the number of events and streams.

## 2.5 Playback of Stored Multimedia Streams

The playback of our active content session objects requires the streaming of stored applet inputs and states. An applet input stream is composed of time-stamped applet events such as state snapshots and inputs. The playback of an applet input stream is accomplished through the re-execution on an applet. Synchronized re-execution of multiple applets recreates a session on the workspace spanned by the applets. The re-execution of inputs to an applet enables interactive reuse of a session and its underlying workspace.

The basic re-execution problem can be reduced to a temporal ordering problem. During capture of a session, inputs and state to each applet are time-stamped and stored as applet input media streams. Since applets are not distributed across workstations, applets and their streams share a common logical time system. Thus, during capture of a session, a full temporal order exists between events across different applet input media streams. To properly schedule events during playback, two pieces of information are needed: (1) the firing time for the event and (2) the event duration (playout time). Because re-execution of applet events is asynchronous and likely to occur on different machines (or even when in the same machine, under variable load conditions), the playout time of applet events is unpredictable. Thus, during the playback of applet events; actual event duration as well as firing time are unpredictable. Consequently, during playback of a session, only a partial temporal order exists between across events from multiple applet streams. We require synchronization protocols that ensure that a full temporal order guarantee exists during playback.

Although the playback problem can now be formulated as enforcing a full temporal order between partially ordered streams, the satisfaction of this goal is constrained by the individual requirements of the applet input streams, such as tolerance to jitter or discontinuities during their playback. This is a different problem than the playback presentation rate problems addressed in [4, 63].

Other important differences limit the suitability of continuous media servers to our requirements. First, there are differences in architectural constraints (for example, see Fig. 2.6) and focus. Research on multimedia extensions for operating systems (see surveys [13, 97, 93]) and for file systems [51, 81, 87, 89, 101] address the support of stored continuous multimedia (e.g., predictive prefetching, batched processing, disk layout, optimal buffering, frame interleaving, etc.). On the other hand, this dissertation focuses on best-effort playback on general-purpose platforms.

Second, research such as [89, 90] focuses on network-centric access model to stored media, designed for the support of a multiple streams, multiple site model. On the other hand, in our domain, a workstation-centric access model is preferable, designed for the support of a multiple stream, single site model. The differences on access model associate different overhead distributions to the media integration problem. In the former access model, variability is primarily due to network latencies. In the later access model, variability is due to three factors: (1) fetching latencies (from the local disk), (2) timer inaccuracy, and (3) asynchronous event playout time. One can deal with fetching latencies via buffering; however, both timer inaccuracies and asynchronous event playout time are more difficult to deal with.

Last, the aforementioned stored media servers do not currently address the integration of heterogeneous media. Media servers often rely on an implicit coupling of media and synchronization that assumes the playback of continuous media only [81] or that heterogeneous media has negligible requirements [93]. The mechanisms found in this dissertation remove awareness of media types from media integration so as to handle heterogeneous media. Steinmetz [96] provided early illustrations of the need for the integration of heterogeneous media. Heterogeneity affects both scheduling and synchronization as shown in [66, 71, 85]. In this dissertation, I present mechanisms for the support of the playback of continuous (synchronous) and asynchronous (discrete) streams.

## 2.6 Scheduling and Synchronization Protocols

The Tactus system [26, 90] supports the replay of integrated media. However, there are important differences. First, the Tactus system has no notion of relative time. Scheduling of asynchronous media is performed *wrt* progress of logical time. Because

(for example, see Fig. 2.6) of timing variability, the relative timing between streams can be lost. Second, Tactus global scheduler uses throttle control over the replay speed of its streams. Its throttle control has two speeds, normal speed (used for replay) or maximum speed (used to catch-up). On the other hand, I rely on smoothed (long-term) throttle controls over the relative replay of a stream. As a result, our replay does not observe abrupt updates (generally, undesirable) to the relative progress of a stream. Finally, the Tactus system depends upon many changes to window systems, operating systems as well as proprietary multimedia device drivers, thus limiting its suitability for general purpose replay as well as its flexibility for handling heterogeneous media streams.

The scheduling of discrete events has also been addressed in computer-based musical systems by Anderson in [2]. However, there are two major differences. First, MIDI events undergo synchronous re-execution referred to as real-time synthesis. Second, the playback of integrated media is not supported.

The playback of stored media requires addressing three basic overheads: (1) fetch, (2) scheduling, and (3) presentation. Unlike with network-based media streaming, in workstation-based streaming, playback overheads (1,2,3) are additive and inter-dependent. Research in network-based streaming, such as [31, 45, 74, 90, 103], relies on adaptive scheduling protocols to manage (bounded) network variability. Their focus is on management of overheads up to the delivery of data to the presentation workstation. These approaches assume processing and presentation on the client workstation incur negligible overheads. Since asynchronous events are particularly sensitive to timing variability, relative inter-media timing can be lost during processing and presentation at the client workstation. This dissertation contains mechanisms that address such timing variability at the client workstation. As a result, our workstation-based mechanisms could be coupled on top of existing network-based mechanisms. Research on management of fetch overheads [87, 89] and scheduling overheads [47, 75] focuses on (low-level) multimedia operating system enhancements for the support of continuous media. Kernel-level manage-

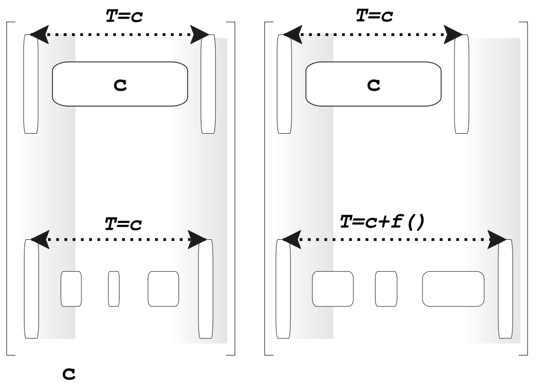


**Figure 2.3: The playback for events from a re-executable content stream is asynchronous and has unpredictable playout duration, much unlike continuous media streams. Unpredictable playout time introduces departures between record timing (left) and playback timing (right), which must be accounted for by scheduling and synchronization mechanisms.**

ment of inter-dependent overheads between competing processes is approached by Mercer in [72]. The mechanisms presented in this dissertation focuses on application-level management and as such, can be built on top of the forementioned multimedia operating system enhancements.

## 2.7 Streaming of Re-executable Content

Applet streams are asynchronous; the actual re-execution time of an applet event or state is unpredictable during scheduling since it is dependent on playback conditions (see Fig. 2.3). We require synchronization policies that handle asynchronous events at a fine-grain of synchronization. The satisfaction of this goal is constrained by the need to preserve smoothness and continuity during the playback of an applet stream. Not all streams have similar tolerances for continuity nor requirements for synchronization. Two issues are central to the streaming of re-executable content: (1) the smoothness or continuity of the playback of a stream and (2) the relative synchronization of the playback of streams in a relationship.

There are systems for the enforcement of full temporal order of fine-grained synchronous or periodical tasks with tight synchronization and strong playback continuity. Some exam-

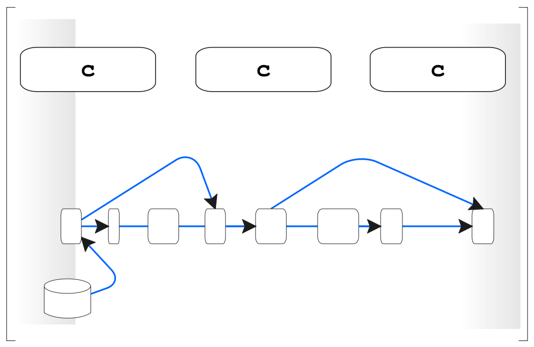


Figure 2.4: Unlike the case of continuous streams, re-executable content streams are composed of statefully dependent events. Statefulness affects the implementation of playback mechanisms. For example, playback (either forward or backwards) from an arbitrary stream event must account for stateful dependencies.

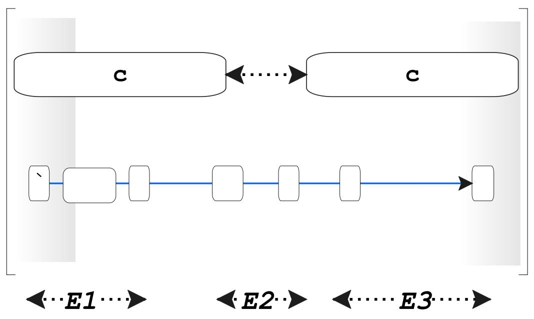


Figure 2.5: A continuous media stream (top row) is divided into frames of expected playout time $c$. The playback characterization of a continuous stream for continuous playback can be *prescribed* by $c$ (e.g., display a frame every $c = 33ms$, present each frame for a maximum duration of $c = 33ms$, control discontinuities between frames to less than $10ms$). Such periodicity is an inherent assumption of many mechanisms for media playback. On the other hand, a re-executable content stream (bottom row) lacks periodicity.

ples are schedulers used for continuous media in general purpose systems [30, 57, 61, 63, 89] and on real-time systems [34, 44, 46, 99]. However, these scheduling algorithms benefit (on one degree or another) from properties of continuous media not found in applet streams. For example, continuous streams have well-defined events with predictable statefulness (if any); neither dropping nor inserting arbitrary events is possible during the streaming of re-executable content (see Fig. 2.4). Similarly, scheduling mechanisms for continuous streams [34, 44] often rely on the fact that the continuous media events (i.e., media frames) have predictable playout times (see Fig. 2.3). On the other hand, because of the asynchronous nature of re-executable content, event playout time is unpredictable.

Some may argue that some continuous media streams such as MPEG video are stateful. For example, consider the following MPEG segment $\cdots M_i = (I_i PBBPBBPBB)\cdots$. An MPEG stream is stateful since the playback of $P$ frames is statefully dependent on neighbor $I$ frames and similarly, the playback of $B$ frames is statefully dependent on both neighboring $B$, $P$, and $I$ frames. However, such statefulness is well-structured (having a pre-determined dependecy structure, predictable length and expected duration). These properties facilitate scheduling and synchronization tasks. For example, consider the algorithms for fast forwarding of MPEG streams in a video server [16] which relies on $I$ frames at variable rates since these lack stateful dependencies, possess deterministic playout time, and have inherent periodicity within the MPEG scheme. On the other hand, statefulness on re-executable content streams is arbitrary between streams as well as events within a stream. For example, a mouse click sequence has an arbitrary size; unpredictable playback duration; and stateful dependencies between events and other streams on the workspace.

Finally, a continuous media stream is divided into frames of expected playout time $c$ (see Figs. 2.5). The playback characterization for continuous media playback can be *realibly prescribed* by such constant $c$. For example, consider that during processing and scheduling, $c$ provides an *a-priori, constant* bound. For example, consider the following constraints: (1) display a frame every $c = 33ms$, (2) present each frame for a maximum duration of $c = 33ms$, (3) control discontinuities between frames to less than $10ms$. Such a-priori knowledge about the predictability and periodicity of event playback is an inherent assumption built-in on scheduling mechanisms for the playback of continuous media (both Schulzrinne [93] and Steinmetz [97] made similar observations). For example, this is particularly true for deadline-based of continuous media [47, 44, 75]. On the other hand,

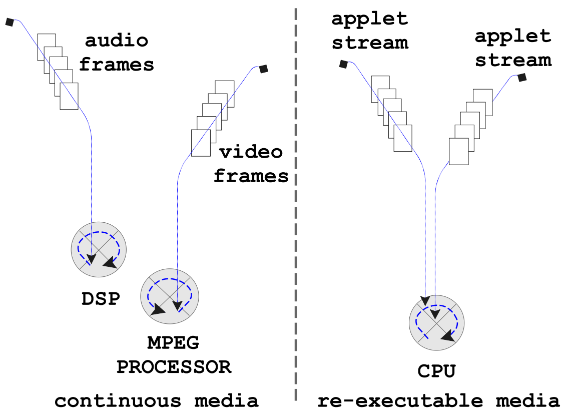


**Figure 2.6: The playback architecture of continuous media (left side) and re-executable content (right side) streams. CM streams are typically dispatched to dedicated processors such as DSPs and MPEG processors. On the other hand, re-executable content streams share processing time on the main CPU.**

a re-executable content stream lacks periodicity. In this dissertation, I provide a playback characterization of re-executable content streams based on the *relative* inter-arrival distributions of sequences of discrete events on a re-executable content stream. This characterization provides a foundation for the integration of heterogeneous media streams.

A sporadic server [46] can be coupled with a deadline-based real-time scheduler [84] so as to handle the integration of asynchronous and continuous media by deadline-based schedulers [44, 97]. The sporadic server is used to handle all asynchronous events with unpredictable playout time. The sporadic server is given a CPU time allotment fraction $q\%$ to process asynchronous events. However, when we consider the streaming of re-executable content, two problems arise with this approach. First, asynchronous events must have a feasible playout time upper bound (that is $q$). Since for a re-executable content stream, playout time is: (1) unpredictable and (2) dependent of the logical definition of events, finding such an upper bound may be impractical. Second, the tradeoff between $q$ and $1-q$ determines the smoothness of the playback of either asynchronous and continuous streams. Ill choices of $q$ affect scheduling fairness and must accounted for in the mechanisms for playback continuity and smoothness.

Many systems enforce full temporal order of asynchronous events with tight synchronization but without continuity guarantees. Examples of these include schedulers used in distributed event simulations [5, 48, 73] and causality preserving protocols such as Birman's

ISIS [8, 9]. Our research problem can be formulated as enforcing playback continuity over an asynchronous event simulation. Several systems enforce full temporal order of asynchronous events with synchronization and continuity but over coarse event grain. These include schedulers in multimedia authoring systems such as Cecelia-Buchanan/Zellweger's Firefly [15] and Kim/Song's Isis [52]. Abstractly, the main difference comes from coarseness of the events they handle. These schedulers deal with high level asynchronous events, more correctly referred to as media parts (such programs and transitions). The coarseness of this grain allows the support of complex temporal relationships. On the other hand, we focus on fine-grained asynchronous events. These events are typically found inside a single media part (such as a program). Both approaches are needed and complementary.

The streaming of re-executable content is also found, most evidently, in the TclCommand stream [42] extension to the CM/T system [89]. Because the CM/T is centered around continuous streams, it lacks the mechanisms for smoothing timing departures during the playback of asynchronous events. Unaccounted timing departures cause synchronization to introduce playback discontinuities to other continuous streams. Our policies and mechanisms for playback smoothness provide means to ameliorate these discontinuities.

## 2.8 Multimedia Modeling

In designing the framework I reused ideas found in the interactive playback of stored continuous streams. To a well-versed reader in multimedia, our framework could be accurately specified as media-independent policies for the playback of relative interval-based scheduling of fine-grained asynchronous events subject to relative timing constraints (both causal and temporal) on multiple $n$-ary relationships that are enforced through pair-wise (binary) synchronization points.

To preserve temporal correspondence on the playback of asynchronous media, we need to rely on relative scheduling. Relative scheduling of stored media was also approached in [86]. To avoid frequent synchronization overheads, I rely on enforcement of temporal intervals through point synchronization. Synchronization points are discussed by Gibbs on [39, 40]. To enforce relationships between multiple streams, I use a modified master-slave model coupled with synchronization handshake. Differing from classical master-slave synchronization (see [40, 96]), I relied on a slave-initiated synchronization handshake. This policy-independent synchronization handshake results in a "*publish-and-subscribe*" model to point synchronization that facilitates relative synchronization across multiple streams.

To specify relationships between multiple streams, I use $n$-ary relationships. These are enforced as multiple binary, point-based, relationships. Work on $n$-ary relationships and their decomposition into binary relationships is also found in the TeleOrchestra [58]. To specify the asynchronous nature of re-executable content, I use a timing departure model, which results in the modeling of imprecise interval-based relationships. The specification of rich and imprecise $n$-ary relationships between temporal intervals has been approached through timed petrinets, most markedly, in the object composition petrinets (OCPN) [30, 61] and time-flow graphs (TFG) [57, 58]. Asynchronous intermissions during playback are modeled in TFG through intermission nodes that buffer the asynchrony between concurrent relative temporal intervals. Gap filtering is used to remove playout gaps; by recomputing a run-time schedule for the media playout. However, scheduling in TFGs lacks the notion of removal of long-term playback trends (which occur when scenarios are played back on workstations of different performance to the record workstation). It is arguable whether zero-gap playback (a short-term optimization) achieves playback smoothness (a long-term requirement). Finally, TFGs use BNF syntax to support very rich formal specification $n$-ary relationships between fine-grained temporal intervals that incorporate asynchronous intermissions as needed in the formulation of collaborative scenarios. However, authoring of the TeleOrchestra rich and imprecise scenarios is currently explicit. Authoring in our model (i.e., of sessions) is implicit. Our specification is transformed at record time, from absolute time schedules into relative time schedules.

To establish long-term controls, I use run-time statistical quality controls (SQC). The delivery of long-term process controls through the use of run-time SQC is widely accepted, led by Deming and Taguchi. However, SQC has been applied mostly to manufacturing lines — lines are sort of streams. My research introduces the use of statistical quality controls for achieving long-term controls over asynchrony and continuity on the playback of multimedia streams. The adaptive feedback mechanism proposed by Venkat-Rangan in [85] can be considered as compensating long-term playback asynchrony trends due to clock drift between record and playback workstations during the playback of stored media. The mechanism is complementary to our work. The media framework on this dissertation extends such notion by introducing the notion of co-existing playback asynchrony trends.

## 2.9 Approach to Workspace Integration

To remove media-awareness from the framework, I introduced plug-in stream controllers, which delivered our notion temporal-awareness to an applet. There are modular streaming architectures for the support of interactive playback such as the one from the Interactive Multimedia Association (IMA). The orchestration of multiple applications in a workspace is also approached in [35], through fine-grain synchronization that relies on exception handling. However, this approach lacks the notion and mechanisms to deal with playback continuity and smoothness. The VuSystem [60] uses a different approach; it uses a data-directed propagation of timing constraints between pipelined modules through back-pressure using a ready/not-ready protocol. However, since each module is only aware of its neighbor timing constraints, the data-directed approach lacks mechanisms for the compensation of long-term playback trends.

Finally, because of its complexity, adding collaboration-awareness to applications usually requires their modification as argued by Ellis [32]. The approach taken in this dissertation fits within this classification. Our collaboration-aware features are delivered through an object class, referred to as the Replayable object class.

## 2.10 Concluding Remarks

The contributions of this dissertation relate to two main areas of research: computer-supported collaborative work and multimedia systems. With respect to collaborative systems, this research relates to shared workspaces and artifact-based collaboration systems. This dissertation introduces a rich content artifact, its features, and a novel paradigm for the support of richer forms of asynchronous collaboration, based on the notion of workspaces as being authored artifacts that can be playback and manipulated. With respect to multimedia systems, this research relates to media integration (i.e., modeling, specification, scheduling and synchronization) of heterogeneous media streams. In particular, this dissertation in addressing the support of replayable workspaces, describes innovative mechanisms for the characterization and integration of the streaming of re-executable content *wrt* continuous media streams.

# CHAPTER 3

# THE PARADIGM OF SESSION RECORD AND REPLAY FOR ASYNCHRONOUS COLLABORATION

*"... and all our knowledge is ourselves to keep."*

— *Alexander Pope. 1688-1744.*

## 3.1 Introduction

Several systems for the support of asynchronous collaboration provide ways to model the interactions among users and the evolution of collaboration repositories [41, 55, 79]. In this chapter, we present a complimentary paradigm for asynchronous collaboration that allows users to record and replay an interactive session with an application. Among us, we may want to refer to this paradigm as a sort of WYSNIWIST (What You See Now Is What I Saw Then) [65]. The paradigm introduces an associated data artifact, the session object, used to capture the collaborative session. Figure 3.1 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. 3.1a). The replay of the session uses the data stored in the session object to recreate the look and feel of the original session (Fig. 3.1b).

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

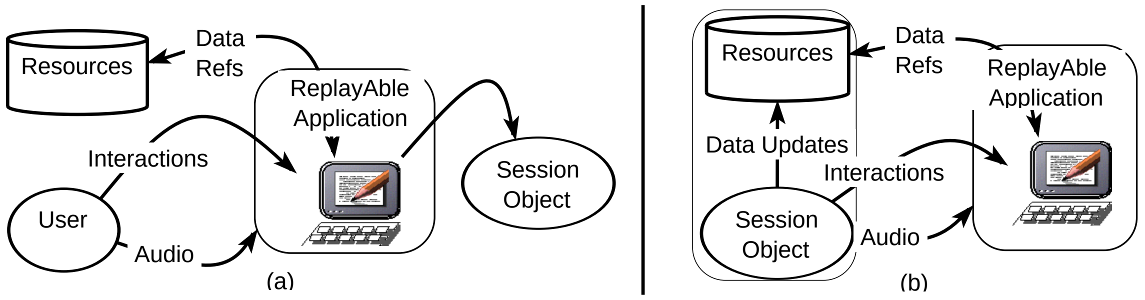


**Figure 3.1: Capture and replay of an interactive session.**

in collaborative work. First, 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. Second, because input events are recorded, session objects are typically small in size and thus easier to exchange among collaborators. Finally, because the application is re-executing the original session, the highest possible fidelity of replay is achieved, which for some domains may be essential.

The rest of the chapter is organized as follows. First, I illustrate some applications of the paradigm. Next, I present the goals of our design. Next, I describe how session objects are modeled. Finally, I discuss design issues in building a system to support the paradigm.

## 3.2 Examples

The next examples illustrate how different asynchronous collaboration scenarios could benefit from both the paradigm and its artifact. Preliminary experience with various prototypes also made us aware of the potential for the following further uses of the paradigm.

### 3.2.1 The Support of Near-Synchronous Collaboration

Our work was originally motivated by the UARC (The Upper Atmospheric Research Collaboratory) project, a collaboratory experiment among domain scientists in a wide-area network [18]. The domain of research among the scientists is space science. From the use of the current version of the UARC system over the past year, it has become clear that all domain scientists are not always able to be present at all times on their workstations to observe the data arriving from the various remote instruments. One reason is that the scientists are often working from different time-zones — the geographical distribution of scientists spans from Denmark to California. Secondly, because the space phenomena being observed are often not well-understood, it is not known *a-priori* when interesting data will be observed. Providing support for some form of session capture and for allowing scientists to exchange annotated session recordings should facilitate *both* asynchronous and synchronous collaboration among them.

### 3.2.2 Using the Artifact as an Exchangeable Document Part

Consider a code walkthrough. It consists of a reader, a moderator, a clerk, and several reviewers. Often, reviewers have different areas of expertise. In fact, most of the time, a synchronous collaboration of reviewers with disjoint areas of expertise is both unnecessary and, in some cases, impractical. The feasibility of a synchronous collaboration approach was shown in the ICICLE system [12]. Although, there are some benefits to holding such a meeting, providing an asynchronous collaboration mode also seems appropriate.

Under our paradigm, the reader role becomes a baseline recording. Each reviewer independently walkthroughs over the code. Reviewers work asynchronously and edit, splice, and annotate segments of the baseline recording with their interactions and annotations.

It is well known that code walkthroughs are not only used for detecting errors. Indeed, they are also intended to share knowledge and to bring people on-board. Since a recorded walkthrough session captures both actions as well as annotations, and it can be replayed at any time, the session object therefore becomes on-line, *live* documentation of system validation.

### 3.2.3 Using the Paradigm to Capture Collaboration Content

In the Medical Collab, we support the type of collaboration that occurs between a radiologist and a doctor over radiographs to diagnose a patient's medical problem. Doctors and radiologists often have very busy schedules, so the ability to collaborate asynchronously is needed.

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' *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.

### 3.2.4 Using the Artifact as an Active Process Description

Consider the use of tutorials. Tutorials typically illustrate how to perform a task — i.e., a process instance. One step further than tutorials is the idea of process capture. The goal of process capture is encapsulate the use and description of a task. Unlike tutorial documents, the paradigm's artifact allows one to encapsulate an *active* — rather than passive — description of a process. That is, rather than just illustrating a task, the session artifact re-executes the task and thus leaves the underlying application in a state that allows its user to continue interacting with the application.

### 3.2.5 Using the Artifact for Process Analysis

Consider the task of building a user interface using a graphical interface builder. By recording the GUI building session, we obtain both: 1) an active document that captures the rationale of why the objects were placed in a given arrangement; and 2) a tutorial that shows and reconstructs the resulting layout and connections. A collection or library of such active artifacts has reuse and knowledge-capture value to organizations. Therefore, means to modify, interact, and browse these session artifacts are needed. This can be regarded as a *digital library* of process descriptions — active artifacts. By supplying own context data to a generic artifact, users may be able to transform a generic process description to a process instance.

## 3.3 Goals

The following are our goals in designing a system that makes an effective use of the paradigm:

- The replay of a session object must be consistent with the original session. This translates to the need to provide a synchronized replay of the data streams and to the need to maintain a consistent view of referenced resource inputs.

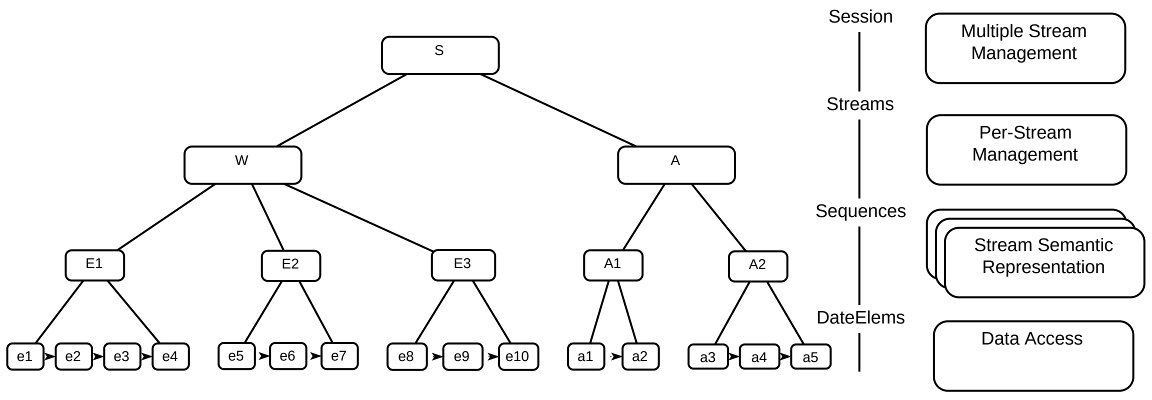


**Figure 3.2: The object hierarchy (a) and corresponding abstraction layers (b).**

- Users must be able to successfully manipulate the artifacts in ways that capture and lead to collaboration. As has been the experience with the use of email for collaboration among a group, we expect that the users will need features such as the ability to exchange, edit, browse, and interact with session recordings. Some of the features above are similar to those found in the VCR metaphor.

- It is desirable for users to be able to statically browse session contents for events of interest. For instance, a user-interface analyst might be interested in browsing through the recording to determine when a particular command was typed or when the mouse was dragged.

## 3.4 Modeling of Session Objects

The session object encapsulates all the information needed to replay a recording and is the building block of this paradigm. The session object is composed of multiple stream objects, such as the streams for audio and window events. Each stream is composed of sequences of data elements, where data elements represent the lowest-level of granularity at which events are captured. For audio, data elements correspond to audio frames. For window stream, the data elements typically correspond to events such as MouseDown, MouseUp, etc. Each object class provides functionality which is used to build services for its parent object class in the hierarchy.

A session object is composed of multiple streams. For example, the session object

$S$ is composed of two streams: $S = \begin{pmatrix} A & W \end{pmatrix}$, where $A$ is the audio stream and $W$ is the window stream. **Streams objects** are composed of multiple event sequences. For example, $W$ is composed of the following three sequences: $W = \begin{pmatrix} E_1 & E_2 & E_3 \end{pmatrix}$, where each $E_i$ represents an abstracted sequence. **Sequence objects** are composed of one or more data elements. For example, $E_2$ is composed of the following three data elements: $E_2 = \begin{pmatrix} e_5 & e_6 & e_7 \end{pmatrix}$, where each $e_i$ represents a data element. **Data element objects** either encapsulate or proxy data items or objects. Figure 3.2 shows the correspondence of the abstraction layers and the object hierarchy. The model of $S$ is given below:

$$S = \begin{pmatrix} W = & \begin{pmatrix} \overbrace{\begin{pmatrix} e_1 & e_2 & e_3 & e_4 \end{pmatrix}}^{E_1} & \overbrace{\begin{pmatrix} e_5 & e_6 & e_7 \end{pmatrix}}^{E_2} & \overbrace{\begin{pmatrix} e_8 & e_9 & e_{10} \end{pmatrix}}^{E_3} & \cdots \end{pmatrix} \\ A = & \begin{pmatrix} \underbrace{\begin{pmatrix} a_1 & a_2 \end{pmatrix}}_{A_1} & \underbrace{\begin{pmatrix} a_3 & a_4 & a_5 \end{pmatrix}}_{A_2} & \cdots \end{pmatrix} \end{pmatrix} \tag{3.1}$$

The **session abstraction layer** provides services for the management of multiple streams. For example, it provides inter-stream synchronization services. The next layer, the **stream abstraction layer** provides per-stream management services. For example, it provides adaptive stream scheduling services to adapt to each stream's performance requirements. Its services provide support for the handling of heterogeneous media such as: *continuous media* and *re-executable content* streams. A re-executable content stream updates the state of the application. An continuous stream is typically used to augment the annotation content of a session. For example, the audio stream (A) is a continuous media stream. On the other hand, the window stream (W) is a re-executable content stream. The following layer, the **sequence abstraction layer** provides efficient sequence-based access to data element objects. This layer groups low-level events into logical units that must be executed as an atomic unit. Consider the following two window events, MouseDown immediately followed by a MouseUp. In this case, this layer abstracts these as $E_1 = \begin{pmatrix} \texttt{MouseDown}[\cdots]\texttt{MouseUp} \end{pmatrix}$, that is, a MouseClick sequence. The **data element abstraction layer**, the lowest layer, provides transparent access to data element objects. These data elements can reside in local disk, remote repositories or be already in memory. Regardless, this layer provides transparent access services to the sequence abstraction layer.

## 3.5 Design

Session objects are similar to video recordings. Both are composed of temporal multimedia streams, both can be used for describing processes, and in both recorded segments can be edited, copied, and exchanged to fit user needs. With the help of the VCR metaphor we hope to facilitate the discussion of the features of the paradigm.

### 3.5.1 Recording a Session

A session with an application can be modeled as interactions with the application and its data resources. To capture the session, we record these interactions. To increase its information content, we also simultaneously record voice annotations. For each of these streams, a per-stream sampling module is provided to efficiently record the events. In capturing interactions, we considered the following issues. Interactions could be captured by means of recording either (1) user-level operations over the application, (2) window events, or (3) display updates. User-level operations (e.g., Open, Print, Quit commands) are at a more abstract level than window events, but require extensive work in making existing applications replayable. Furthermore, certain operations such as gestures using mouse movements are typically not captured. Both approaches (2) and (3) allow capturing of mouse-movements used for gesturing or for indicating hesitation on the use of a feature of the application. We, however, decided to record window events, rather than display updates. While the use of display updates is application-independent and requires less sophisticated synchronization schemes, it has the disadvantages of a larger session object size, the inability to query the contents of a session object, and the inability to interact with a session object — features of collaborative interest which are possible with the use of window events. Although we are currently exploring approach (2) (window events), we feel, however, that a complete system would give the user the option to also record display updates.

Replay support using approach (2) requires capturing the state of the application. We require that the application provides functions to record and reset its state. To record a session, the toolkit calls back the application to tell it to capture its state. Resource references (e.g., environment, files etc.) must also be faithfully reproduced during replay. This problem is addressed in the Section on Replaying a Session.

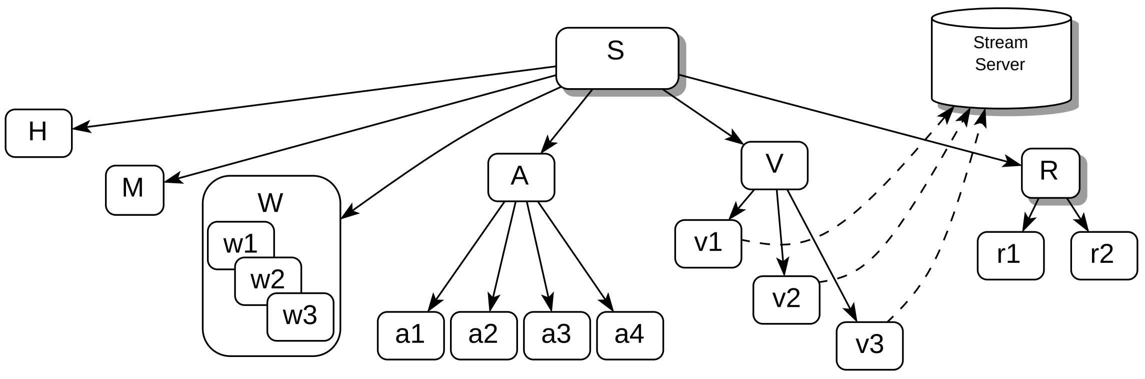


**Figure 3.3: Storage representation of recorded sessions.**

Each stream is represented as a tuple containing the stream's initial state and its events. For example, the initial state of the window stream contains the state of every object of every window. This is accomplished by periodically sending a write message to the root parent object of every window in the display hierarchy. We use $S_t$ to denote the state checkpoint at time $t$.

### 3.5.2 Storage and Access of Sessions

Session objects are persistent objects and must be stored on disk. However, in order to be exchanged and be used over time, we need to provide: (1) an editable representation for them and (2) efficient access to them.

To address (1), we opted for a file-based representation for session objects, as shown in Fig. 3.3. A session object is stored as a directory $S$. To illustrate this representation, suppose that $S$ consists of a window stream ($W$), an audio stream ($A$), and a shared video stream ($V$). It is composed of a session header file $H$, a measurements file $M$ containing the data needed to support synchronized replay of the session, a header file for each stream ($W, A, V$), and a resource directory $R$. Each stream maps to a file. However, the stream data may be stored directly in the header file (as for $W$), indirectly as references to persistent objects (as for $A$), or as proxy references to shared objects (as for $V$).

Streams typically have different access requirements. To address (2), the use of this file-based representation allowed us to tailor access strategies to each stream's requirements. To amortize read access costs, we used prefetching of events. To amortize write access costs, we used buffering of events. These techniques were optimized so as to balance the overhead for disk-accesses vs. the available time for stream-execution. Note that access and execution tasks execute and compete for the same resources within the same CPU.

### 3.5.3 Replaying a Session

To ensure a faithful replay, we addressed the following questions:

- *Is there a need for synchronization?* Yes. Early on, our experiments showed that both streams (audio and window) had different susceptibilities to load conditions and that their rate of progress was dependent on the current load. Therefore, the ability to dynamically adjust the speed of replay of streams is desirable.

- *How to synchronize different streams?* We decided to test several different protocols for synchronizing audio and window streams. Two way protocols, which maintain relative synchronization between the two streams at the cost of their occasional re-synching, led to audio discontinuities, and were deemed untolerable by users. Consequently, we designed a one-way inter-stream synchronization scheme that synchronized slave streams to a master stream. The scheduling of events from a slave (window) stream was periodically adjusted to synchronize to its master (audio) stream.

- *Is an adaptive synchronization protocol needed?* Yes. The variances due to CPU availability, DMA access, thread overheads, disk access, reliability of timing services, etc., affected the scheduling of both window and audio streams. Our results in [66] showed that an adaptive protocol that attempts to compensate for varying load generally performs better across all load conditions.

The issues are explore in detail in Chapter 4.

Streams execute as cooperating thread tasks in a single CPU. The infrastructure provides two generic thread models. Figure 3.4(a) shows the thread model used to replay window events. On the average, during the sampling of the window stream, between 10 to 30 window events per second are generated by the user. However, during replay, these events must now be produced *and* consumed by the application itself. Therefore, a producer and consumer thread pair is used. 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 the protocol used for multi-stream synchronization. The consumer thread gets events from the shared queue and dispatches them to the window system for

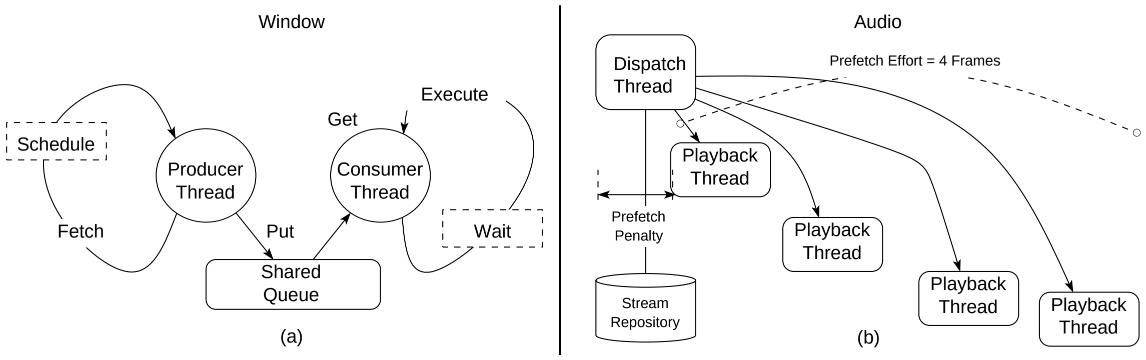


**Figure 3.4: Thread models for replay of the window event stream (a) and for the replay of the continuous audio stream (b).**

event replay. Figure 3.4(b) shows the thread model used to replay audio frames on the NeXTs. Read access for the audio stream relies on a parametrized disk-prefetching of audio frames.

The replay also introduces problems with the handling of resource references (e.g., fonts, files, devices, etc.) made during the recording of a session [17]. On the replay platform, referenced resources may be unavailable and, even worse, if available, may not be in the same functional state. To address the unavailability problem, resource references are classified as being public or private. Public resources are assumed to be widely available across platforms. Private resources need to be made available to replay platforms. To address the state problem, resource references are also classified as being stateful or stateless. Stateless resources are made available only when classified as private resources. Stateful resources must be available under a consistent state to replay platforms. A session recording also contains a *resource requirements list* and a *resource shipping list*. The resource requirements list indicates which resources are referenced by a session. The resource shipping list indicates which, how, and where resources referenced by a session should be accessed. These lists are provided at replay time to ensure a correct replay of the recording.

### 3.5.4 Editing of Sessions

The editing problem comprises recording new streams over a baseline recording, copying and pasting stream segments, extending a session with additional interactions and annotations, and the like. However, the editing problem of re-executable content is a difficult one. For applications domains where state checkpointing is feasible and robust, we

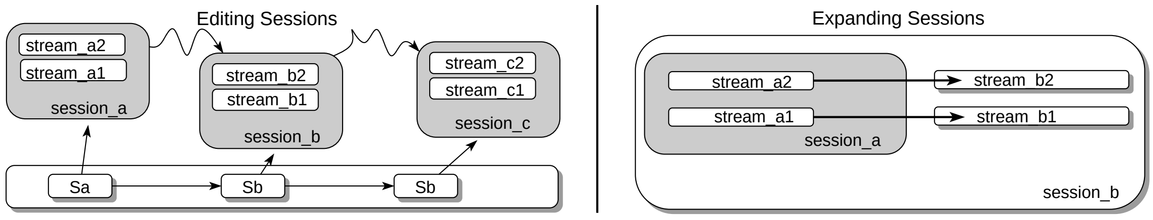


**Figure 3.5: Editing (a) and expanding (b) session objects.**

have currently explored some preliminary approaches to this problem.

On a VCR, streams are functionally independent and editing is typically done on a per-stream basis. We take a similar approach. However, in the case of session recordings, editing can only be done between well-defined points across all streams. Suppose that we want to dub-over a segment of the window stream. Conceptually, this is equivalent to replace some stream segment modeled by some events $[e_i, .., e_{i+k}]$ with a new set of events $[e'_i, .., e'_{i+n}]$. However, we must address the following two constraints. First, editing must preserve the synchronization that exists between streams. Secondly, since streams are not stateless — events have to be executed in the correct state — editing must preserve correctness of replay.

To maintain the correspondence that exists between streams, editing must be performed with identical sampling and synchronization schemes as in the original recording. To ensure correctness of replay, a stateless execution boundary is needed. One strategy is to allow only editing to start from a state checkpoint. The efficiency of this editing strategy depends on how far apart the state checkpoints are from each other. Note, however, that if the replay is based on display updates instead (previously discussed in the Section on Replaying a Session), editing of a session becomes much simpler since the streams are now stateless.

Figure 3.5 shows two of the potential ways of editing sessions. Figure 3.5a models editing of a session through concatenation of previously recorded session segments, $s_a$, $s_b$, and $s_c$. Figure 3.5b models a session $s_b$ that extends a previously recorded session $s_a$ with additional interactions and annotations. In Chapters 6 and 7, I revisit the problem of editing a session in terms of workspace modeling (as in Section 6.8) and data modeling (as in in Chapter 7, Fig. 7.4).

### 3.5.5 Browsing a Session

Use of the VCR metaphor seems appropriate for doing simple sequential search of a recording. A VCR has two modes of forward search: fast forward and fast replay. A VCR easily performs any of these operations because it is based on a stateless model. In our case, streams are associated with a state but we show next how these operations can still be done efficiently. Say we want to fast forward from a current event $e_i$ to event $e_n$. We can not just randomly index to that event because: (1) event $e_n$ may have some causal dependency with a previous event in the sequence; (2) event $e_n$ may not provide a clean execution boundary across all streams.

There are two solutions to these problems: (a) replay all events in the sequence $< e_i, ..., e_n >$, but at a faster (perhaps variable) rate or (b) jump to the last state checkpoint $S_t$ prior to $e_n$ and then apply the sequence of events $S_t + < e_j, ..., e_n >$. Solution (a) (i.e., fast replay) is reasonable when the forward distance is small. Solution (b) (i.e., restartable replay) is appropriate when the forward distance is large. Note that in general, variable speed replay relies on strong synchronization support, requiring the scaling of the rate at which events are to be dispatched while maintaining the relative synchronization between streams (or requiring the disabling of replay of some of the streams such as audio).

Backward replay can be implemented if every event has an inverse or undo operator. For many streams (e.g., discrete streams), the ability to execute the stream in backward order is likely to be difficult. In such cases, backward replay may have limited or no feasibility.

### 3.5.6 Interacting with Sessions

Interactions with a recorded session can be performed at two granularities: (1) between recorded sessions and (2) within a recorded session. Suppose that session $s_1$ results in the drawing of a layout and session $s_2$ results in the formatting and printing of a layout. A user may wish to replay session $s_1$, add some final touches to the layout, and then print it by means of session $s_2$. This is an example of between-type interactions. Interactions within a session are possible through the use of the resource reference list. In this case, users parametrize and modify resources referenced to by a session to fit their needs and requirements (e.g., printing a different file than that printed in the original recording by replacing the file resource).

### 3.5.7 Other Features

Users, such as interface analysts, may want to browse a session recording for interesting events.

Static browsing of the session contents is a feature that does not have a simple match in the VCR model. Consider a window stream segment corresponding to having a user click on a window, position the cursor and then start typing the word "*password*". Such a content can be potentially retrieved from the window stream without having to replay the session. Knowledge discovery tools can be created to examine and peruse repositories of these digital recordings.

Users also need a way to efficiently exchange session objects. Resources referenced by a session must be faithfully forwarded or equivalenced during replay. Mailing of a session $S$ reduces to the problem of mailing of directories (recall example in Fig. 3.3). The mailing of a shared stream, such as $V$, is straightforward, by means of using relative referencing to the repository $R$. The resource reference lists and resource shipping lists are used here to ship resources.

## 3.6 Concluding Remarks

In this chapter, I presented a new paradigm and its associated collaboration artifact for the support of asynchronous collaboration.

The paradigm and its underlying synchronization infrastructure allow users to capture interactive sessions with an application into data artifacts, i.e., the session objects. Unlike other data artifacts, session objects are active objects. Session objects can be manipulated, replayed, interacted with, and analyzed to fit the needs of collaborators.

Sessions objects are represented by temporal streams, kept synchronized during the replay of the session. Furthermore, this new data artifact introduced by the paradigm can also be used to enrich the artifact chest of existing multimedia authoring and collaboration systems, so as to capture intra-task descriptions in both asynchronous and synchronous collaborative systems.

# CHAPTER 4

# DEALING WITH TIMING VARIABILITY IN THE PLAYBACK OF INTERACTIVE SESSION RECORDINGS

*"The farther back you can look, the farther forward you are likely to see."*

— *Winston Churchill.*

## 4.1 Introduction

The "*replay by re-execution*" approach requires addressing *timing-variability* during re-execution of asynchronous events. Timing variability may be due to a number of differences between playback and record workstations that affect re-execution performance; for example. the workstation's processing power, load condition, and accuracy (or inaccuracy) of timing services. Whereas some of these performance differences such as varying load conditions have unpredictable time-spans, others such as the inaccuracy or biasing of timing services are predictable and when present, introduce long-term asynchrony trends on the playback. Measures are needed to compensate for both long-term and short term playback asynchrony trends. This chapter focuses 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. In the experiments described in this chapter, I relied on session objects composed of a re-executable content stream (the window stream) and a continuous media stream (the audio stream). During replay, audio playback must maintain relative timing *wrt* the playback of user interactions. Since the playout time of asynchronous events is unpredictable, such synchronization must be performed *wrt* the relative progress of the streams (as opposed to *wrt* the progress of global schedule time.

However, the re-execution of events is subject to timing variability. The window stream, as a re-executable content stream, is composed of stateful, aperiodical, discrete, and asynchronous events. The continuous audio stream, on the other hand, is stateless, periodical, continuous, and synchronous stream (refer to Chapter 2, Figs. 2.4, 2.3, and 2.5). Adaptive strategies for integrated replay of stateless, periodical and continuous media can not efficiently address requirements of the window stream. For example, dropping/duplicating frames, a strategy commonly used for continuous media such as video and audio, cannot be usually allowed). This chapter describes scheduling and synchronization mechanisms for the support of faithful replay of session objects.

The rest of the chapter is organized as follows. First, I present the goals and requirements of session objects. Next, I discuss modeling and design issues. Then, I compare the performance of several synchronization protocols and analyze my findings. Finally, I present my concluding remarks.

## 4.2 Goals and Requirements

The UARC and the Medical Collab domains illustrate the need for transportable objects, suited for replay across similar workstations. The support of these domains have the following requirements:

**R1:** Support faithful replay of a session. In particular, it is necessary to compensate for timing variability.

**R2:** Reduce reliance on hardware and operating systems support, so as to reach a broad collaboration base.

This chapter focuses on timing variability issues in the re-execution of discrete, functional streams in the replay workstation. Research issues addressed include:

- integration of re-executable content streams (i.e., stateful, aperiodical, discrete, asynchronous streams) *wrt* continuous media streams.
- 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.

To address timing variability, a new algorithm for run-time, adaptive scheduling for the support of integrated media replay is presented. 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. The inter-stream asynchrony floats under statistical control as a function of the scheduling interval.

## 4.3 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.

The proposed solution to this problem has two parts:

- a synchronization mechanism and
- an adaptive scheduling mechanism.

First, the synchronization scheme and its operation is described. Then, the adaptive scheduling mechanism and its derived scheduling algorithm are analyzed. Figure 4.1 provides an intuitive look to our adaptive scheduling mechanism.

### 4.3.1 Asynchronous Media Streams

The prototype used in this experiment was a monolithic application, a McDraw-like object-oriented drawing application, with temporal-awareness of two streams, window and

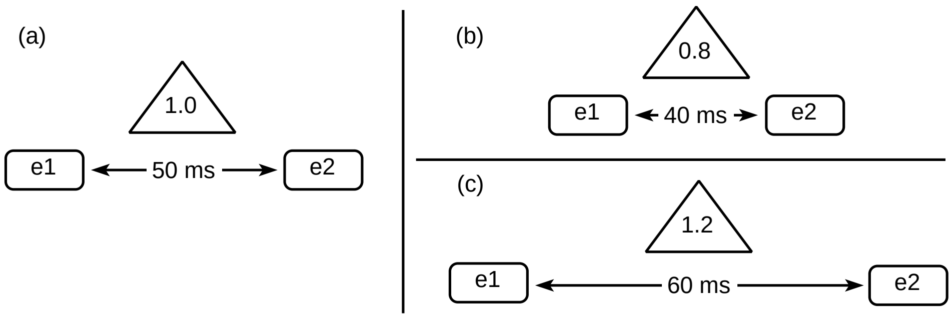


Figure 4.1: Intuitive look at our adaptive scheduling mechanism. Part (a) of this figure shows the inter-event delay $\Delta_{1,2} = 50ms$ between events $e_1$ and $e_2$. Our approach adapts inter-event delays (i.e., the idle time between events) to compensate for trends in asynchrony. Adjustments are 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.

audio streams. Both window and audio streams were modeled as discrete, asynchronous streams, however, of 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:

$$t(e_i) = E(e_i) + f_o(e_i) \tag{4.1}$$

$E(e_i)$ 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, pre-emption, timing services, context switches, etc., and

**A3:** 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, pre-emption, context switches, CPU scheduling, etc..

The playback of a session object on a workstation requires 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 chapter.

### 4.3.2 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 Chapter 10). 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, pre-emption, and large system call overheads while lack peer-to-peer guaranteed scheduling time. Our synchronization precision was originally targeted to support about 0.100 to $.500ms$ for example, that of audio-annotated slide shows as quoted from [13]). Finer grain synchronization would have imposed demands requiring OS-level support [13], thus compromising our goal for high level support (R2). However, note that this target was originally set for NeXT workstations running Mach OS 3.2 but under Sparc 20/Ultra workstations running the Solaris 2.5.1 OS, I have encountered that the range for current workstations and high-end desktop systems can be described from 0.025 to $.250ms$.

### 4.3.3 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 [96]. 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 [57, 61], our synchronization specifically preserves the relative synchronization relationship ($e_i$ *same* $a_j$) between window and audio streams. Figure 4.2 shows the use of a synchronization event $s_i(A \leftarrow W)$, where $A$ is the audio stream and $W$ is the window stream.

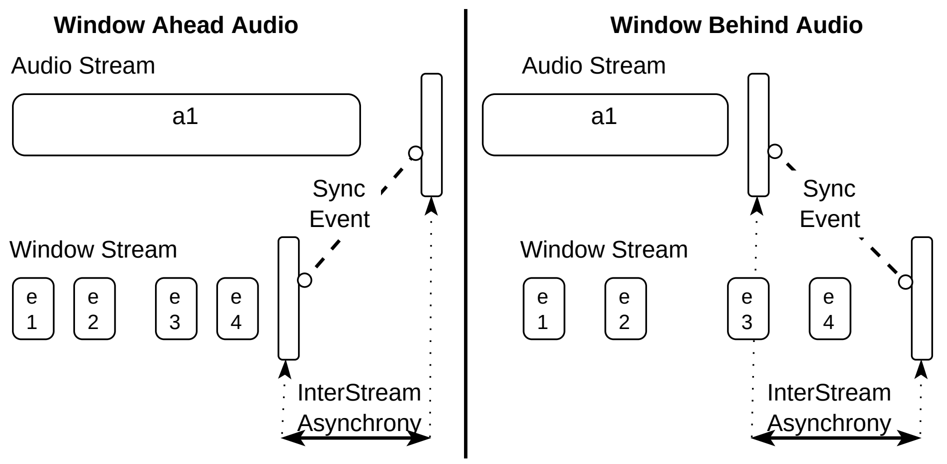


**Figure 4.2: The two inter-stream asynchrony cases. A synchronization event, $s_i(A \leftarrow W)$, preserves relative synchronization between our $A$ (audio) and $W$ (window) streams. The inter-stream asynchrony between corresponding synchronization events is variable. It is estimated by the subtraction of observed schedule times for synchronization events at each stream — e.g., $\epsilon_i = t(s_i(A)) - t(s_i(W))$.**

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., $\epsilon_i = t(s_i(A)) - t(s_i(W))$.

### 4.3.4 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 [13, 39, 96]).

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 [90]). In our prototype, the audio

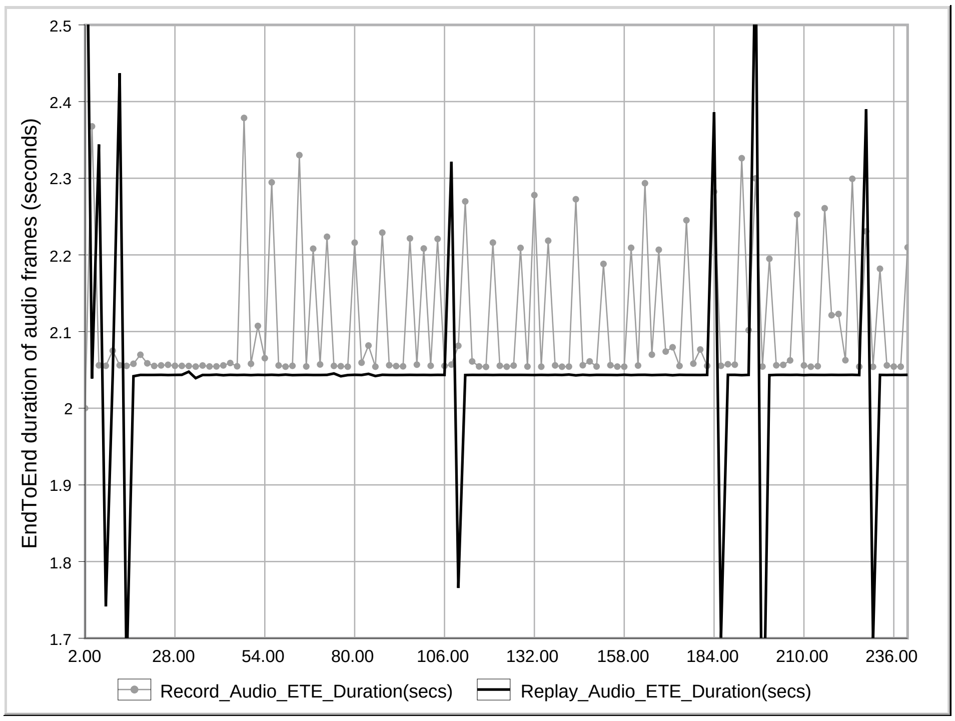


**Figure 4.3: Timing variability observed during record and replay of audio frames. Plots show the end-to-end duration $t(a_i)$ of audio frames during record and replay of a $250sec$ test session. Variability $f_o(A)$ present during record is due to (1) sampling to secondary storage and (2) overheads resulting from the thread/per-frame approach used in NeXTs. Variability $f_o(A)$ present during replay is controlled through (1) prefetching of frames and (2) slave-initiated synchronization measures.**

stream is used as the master stream and the window stream is its synchronizing slave — the window stream schedule is adjusted to preserve synchronization *wrt* audio stream's schedule. 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 4.3 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.

### 4.3.5 Synchronization Policies

In [96], synchronizing operations were characterized by:

- the involved partner(s):
  that is, to which stream to synchronize. The experiments described here implemented the synchronization relationship $(A \leftarrow W)$.
- the type of synchronization:
  that is, whether to use a non-blocking, 1-Way or 2-Way blocking protocol.
- the release mechanism:
  that is, 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. 4.2:

- window ahead of audio condition:

  the window event stream reaches its synchronization event before the audio stream (right side of Fig. 4.2).

- window behind audio condition:

  the window event stream reaches its synchronization event after the audio stream (left side of Fig. 4.2).

Appendix E specifies the various protocols considered in the experiments described through this dissertation 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 [40]. Protocol P3 is our adaptive scheduling algorithm. Protocol P4 is a two-way, stop and wait, protocol for relative synchronization of discrete streams.

### 4.3.6 Adaptive Scheduling Mechanism

The basic idea behind our adaptive mechanism was illustrated in Fig. 4.1. Next, we generalize and 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 (e.g., as in [62]). 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.

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

$$T = t(e_1) + \Delta_{1,2} + \cdots + t(e_n) + \Delta_{n,s_1} \tag{4.2}$$

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

$$T = E(e_1) + f_o(e_1) + \Delta_{1,2} + \cdots + E(e_n) + f_o(e_n) + \Delta_{n,s_1} \tag{4.3}$$

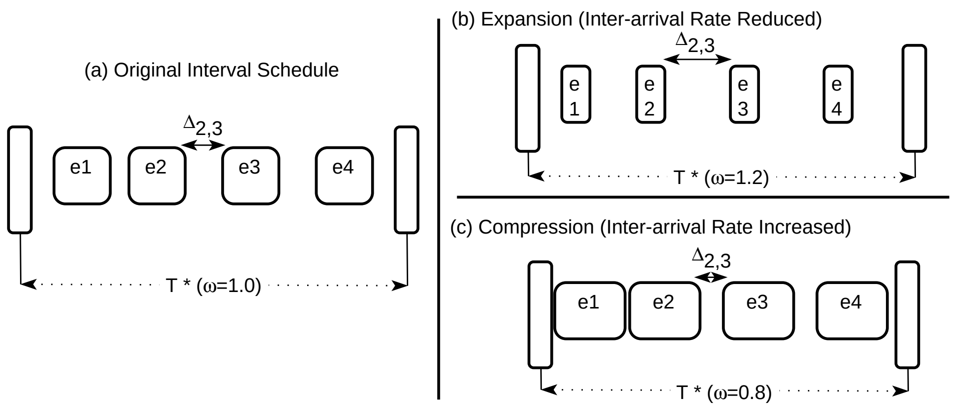

(a) Original Interval Schedule
$\Delta_{2,3}$
e1
e2
e3
e4
T * (ω=1.0)
(b) Expansion (Inter-arrival Rate Reduced)
$\Delta_{2,3}$
e 1
e 2
e 3
e 4
T * (ω=1.2)
(c) Compression (Inter-arrival Rate Increased)
$\Delta_{2,3}$
e1
e2
e3
e4
T * (ω=0.8)

**Figure 4.4: 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.**

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

$$T^* = E(e_1) + \Delta_{1,2} + \cdots + E(e_n) + \Delta_{n,s_1} \tag{4.4}$$

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

$$\epsilon^* = T - T^* = \sum f_o(e_i) \tag{4.5}$$

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 departure — to timing variability during replay at a workstation. The basic mechanism was previously illustrated in Fig. 4.1. In Fig. 4.4, 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 D). Consequently, we obtain a *$\omega$-modified scheduling interval duration* $T_\omega$ formulated as

$$T_\omega = E(e_1) + f_o(e_1) + \omega\Delta_{1,2} + \cdots E(e_n) + f_o(e_n) + \omega\Delta_{n,s_1} \tag{4.6}$$

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,

$$\epsilon_\omega = \sum f_o(e_i) - (1-\omega)\sum \Delta_{i,i+1} \tag{4.7}$$

which by equation (5) can be rewritten as

$$\epsilon_\omega = \epsilon^* - (1-\omega)\sum \Delta_{i,i+1} \tag{4.8}$$

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, 39, 63, 89]). 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.

### 4.3.7 Stream Scheduler(s)

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

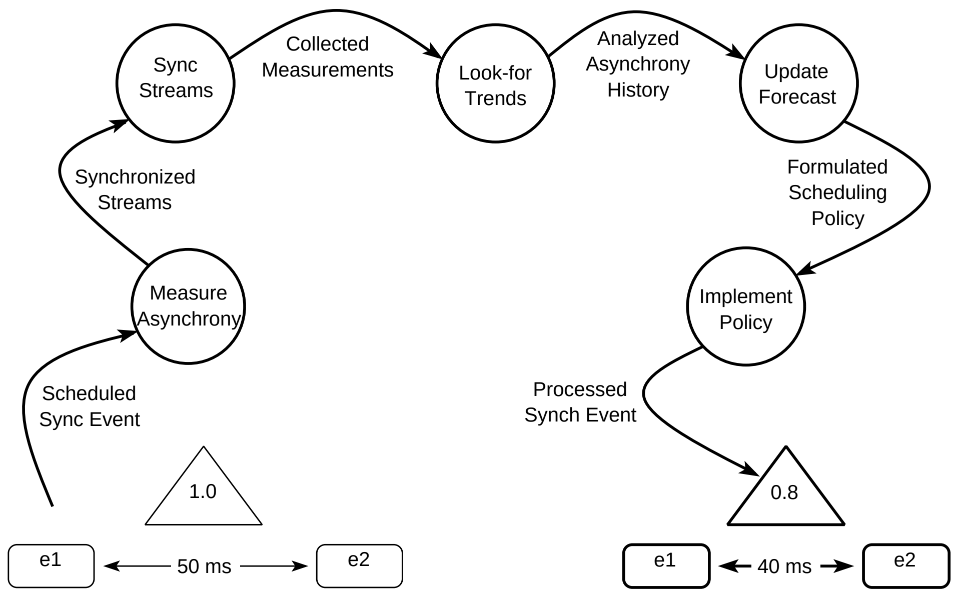


**Figure 4.5: Overview of the per-stream adaptive scheduling process. The $\omega = 1.0$ compensation is revised to $\omega = 0.8$, resulting in 20% smaller inter-event delays for events in the next scheduling interval. A synchronization event may trigger a five phase analysis cycle to measure, synchronize, analyze, forecast, and prevent further inter-stream asynchrony.**

Our scheduling is statistical. The behavior of the adaptive algorithm is specified as follows. If the current asynchrony is large enough, the asynchrony history is examined to determine the presence of trends *wrt* ($1\sigma$) control limits and ($2\sigma$) warning limits — where $\sigma$ is estimated by the stream tolerance $T$ since a $3\sigma$ control limit was found to be too conservative in its reaction to the asynchrony. 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. 4.4. 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),

| 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 4.1: 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.**

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 an attempt to reset the inter-stream asynchrony to zero. The actions carried out in Step 3 are further detailed in Fig. 4.5. 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. 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 $15ms$.

Figure 4.6 provides an intuition about the variability component introduced by differences between various scheduling approaches. Panel (a) shows the original record schedule for an asynchronous event: $e_1 : [t*(e_1), \Delta(e_1)]$; panel (b) shows the uncompensated replay schedule, subject to timing departure: $e_1 : [t(e_1), \Delta(e_1), f_o(e_1)]$; panel (c) shows P2: event (fine-grain) based compensation during replay; $e_1 : [t(e_1), \Delta(e_1) - f_o(e_1)]$; and panel (d) shows P3: interval (coarse-grain) based (adaptive) $\omega$-compensation during replay;

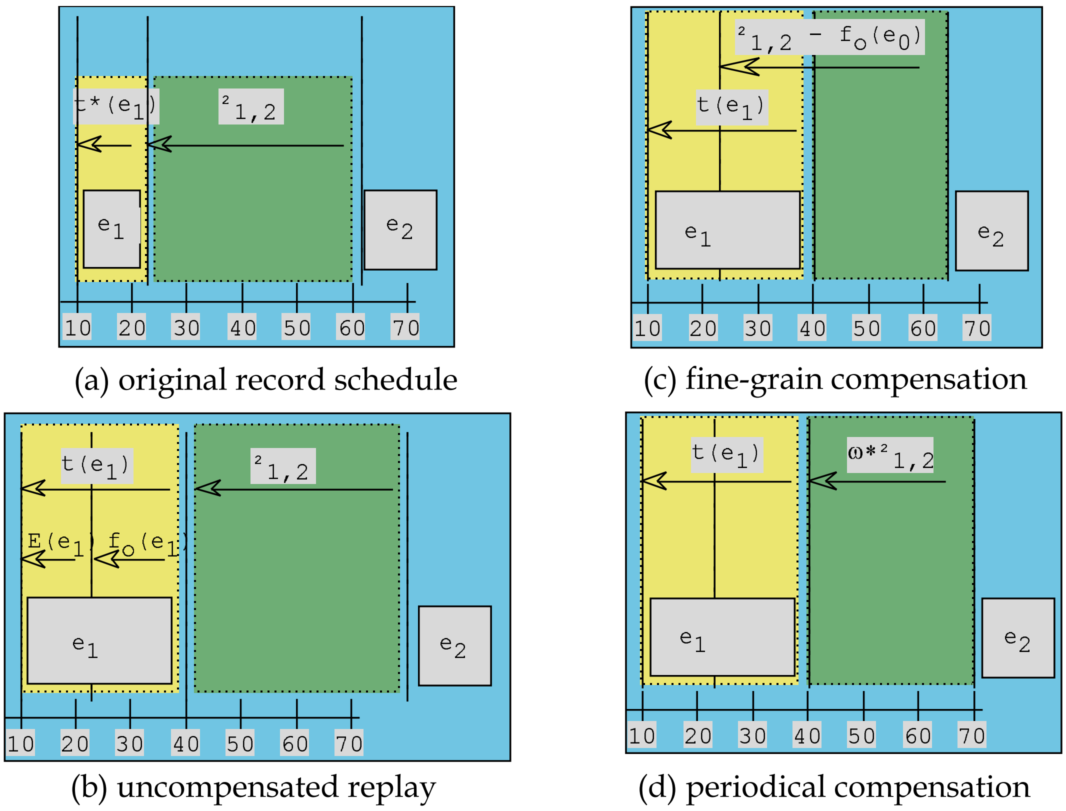


**Figure 4.6: Scheduling approaches: (a) original record schedule for an asynchronous event; (b) uncompensated replay schedule, subject to timing departure; (c) fine-grained based compensation during replay; and (d) interval-based $\omega$-compensation during replay.**

$e_1 : [t(e_1), \omega * \Delta(e_1)]$.

Table 4.1 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). Recall that whereas P3 is adaptive P2 is not. Fig 4.7 shows the corresponding ANOVA for the results presented in Table 4.1.

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.

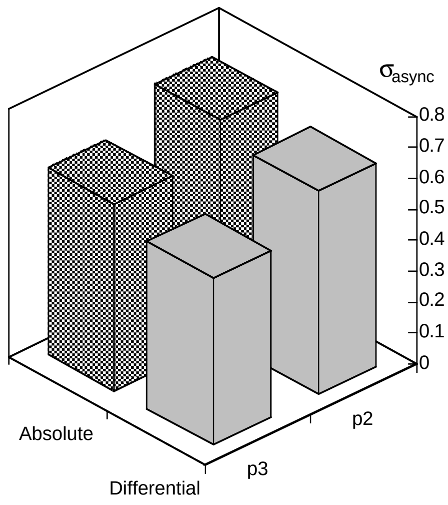


Figure 4.7: Graphical ANOVA of possible scheduling strategies. Differential scheduling is statistically better than absolute scheduling since it reduces reliance on timing services to just synchronization events. Adaptive scheduling is better than scheduling against a fixed schedule since it removes assumptions about the progress rate of a stream.

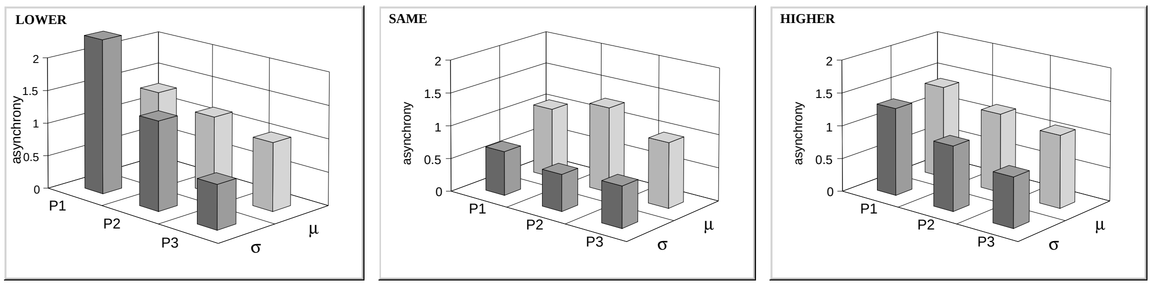


Figure 4.8: Comparison of mean (front row) and variance (second row) on inter-stream asynchrony for protocols P1, P2 and P3. Desirable characteristics are low mean and variance, across all load conditions. Protocol P3, based on our adaptive scheme, meets these requirements. Replay results shown for lower (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.

## 4.4 Comparative Analysis

The protocols described above were analyzed under different background load conditions for the parameter values (discussed on Chapter 10), (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 replayable object class toolkit, a MacDraw-like drawing application was made replay-aware through our toolkit. Using a baseline background load, a 4-minute 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). The load was measured 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). The results of this testing for protocols P1, P2, and P3 are shown in Fig. 4.8. 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 80% and 110% of the original scheduling interval — 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

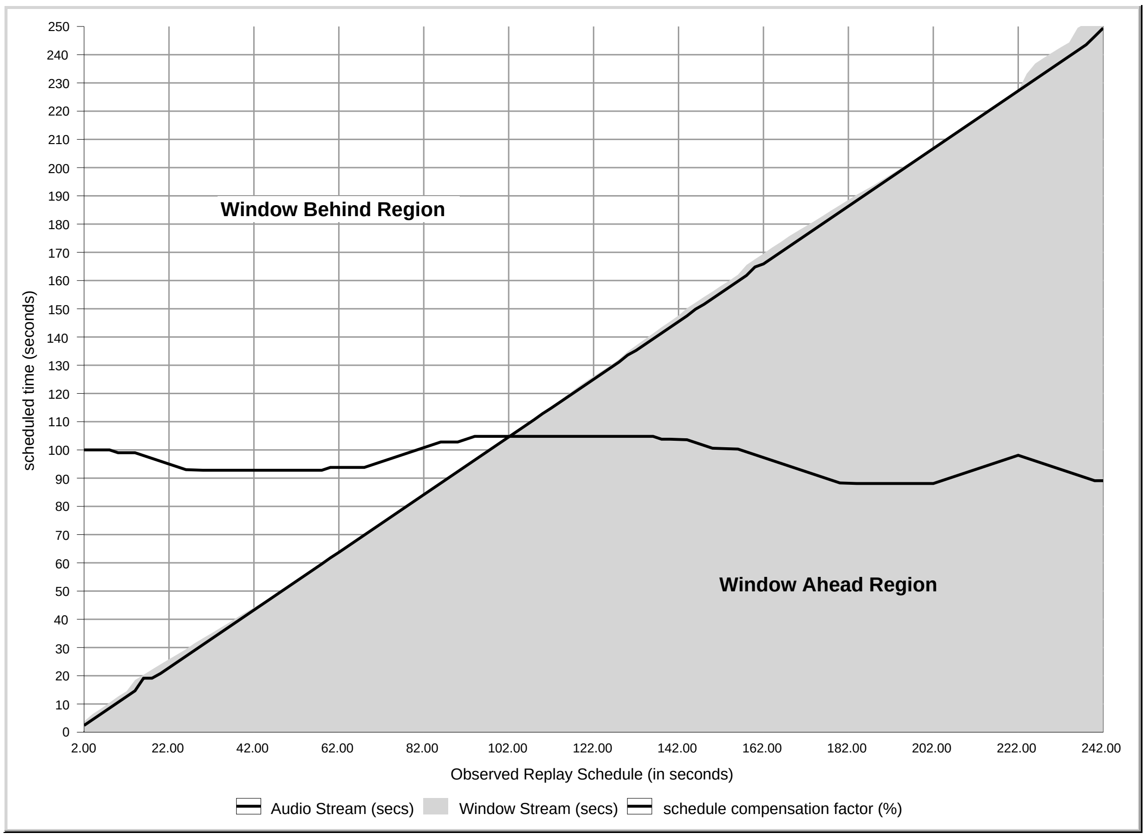


**Figure 4.9: Detailed time-line performance of protocol P3 under higher load conditions for the same session object. Shaded area represents replay schedule for the window stream, diagonal represents replay schedule for the audio stream, and horizontal line represents compensation factor $\omega$.**

for 40 ms. 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 4.9 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

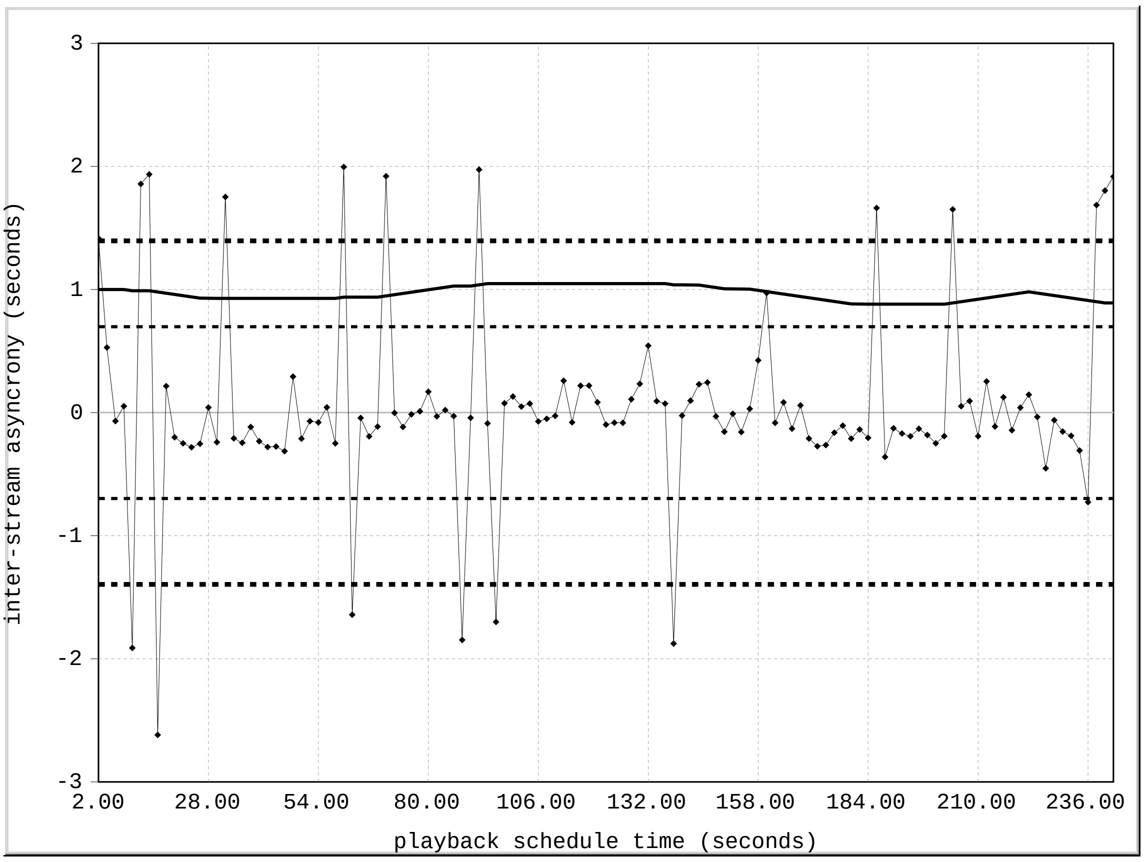


**Figure 4.10: The matching SQC/SPC plot for protocol P3 under higher load conditions for the same session object. On each scheduling interval, the compensation factor $\omega$ (the solid line curve fluctuating around (0.80 and 1.20)) between adapted to trends in the measured asynchrony (the curve of connected diamonds fluctuating between 1 and $-1$).**

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.

### 4.4.1 Analysis of the Results

Some interesting conclusions (refer to Table 4.2) can be drawn for our protocol P3 *across all load conditions.*

| 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 4.2: 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 $\mu_T$.**

- The average asynchrony $\mu_{async}$ for protocol P3 was about half the mean scheduling interval duration — the scheduling interval duration was defined by equation (2). $\mu_T$, ($\mu_{async} \rightarrow \frac{\mu_T}{2}$).
- 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 is 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 4.3.1). 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. 2-way adaptive protocols are examined in Chapters 7 and 8.

## 4.5 Concluding Remarks

This chapter addressed media scheduling and synchronization issues for the support of faithful re-execution of session objects when subject to timing variability. In the prototype used for the experimentation described in this chapter, 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, 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, I implemented a modified master/slave model. Synchronization was initiated by the slave streams.
- To bound inter-stream asynchrony, I used periodical synchronization.
- To remove dependencies and overheads from timing services during a scheduling interval, I used differential scheduler(s).
- To control variability within a scheduling interval, I 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, I adjusted idle schedule time (in the form of inter-event delays).
- To allow adjustments to multiple slave streams, I implemented a scheduler per stream, running malleable time.
- Finally, to integrate these multiple stream schedules, I implemented relative inter-media synchronization, as opposed to absolute media timing.

The adaptive algorithm P3 presented in this chapter 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.

# CHAPTER 5

# REPLAY-AWARENESS: THE NOTION OF REPLAYABLE APPLICATIONS AND WORKSPACES

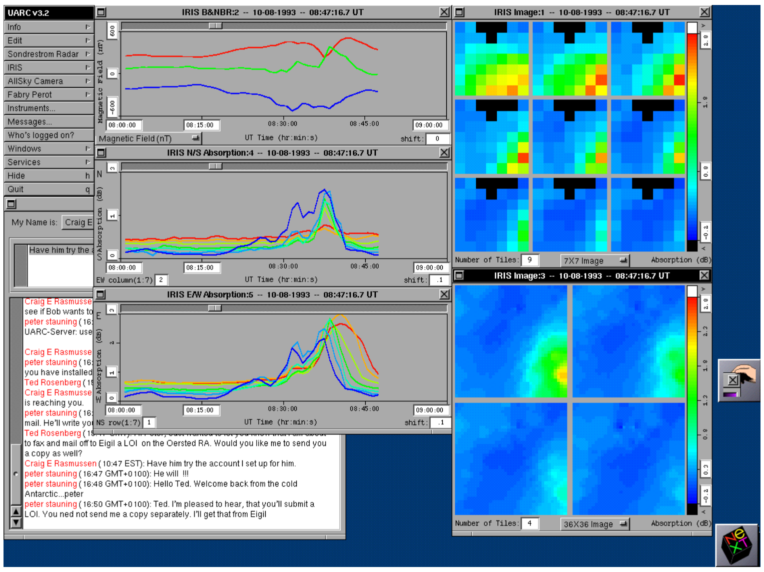
*"I submit to the public a small machine by my invention, by means by which you alone may, without any effort, perform all the operations of arithmethic, and may be relieved of the work which has often times fatigued your spirit."*

— *Blaise Pascal, 1623-1662.*

## 5.1 Introduction

This research was originally motivated by the UARC project (see Fig. 5.1) a collaboratory experiment among space scientists in a wide-area network [19]. 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. As part of an NSF grant (a Medical Collaboratory Testbed), we applied the session record and replay techniques to support collaboration between a radiologist and a doctor over radiographs to diagnose a patient's medical problem.

The goal here is to allow both radiologist or doctor to record a session containing typical user interactions of interests to radiologists and doctors. For example, interactions such as side-by-side analysis of two or more images, gesturing on specific areas of interest on an image, using audio to explain or raise questions about regions of interest in images, and adding text or graphical annotations.



Have him try the a

Craig E Rasmusse
see if Bob wants to
peter stauning ( 16:
UARC-Server: use

Craig E Rasmusse
peter stauning ( 16:
you have installed
Ted Rosenberg ( 15
Craig E Rasmusse
is reaching you.
peter stauning ( 16:
mail. He'll write yo
Ted Rosenberg ( 15
to fax and mail off to Eigil a LOI on the Oersted RA. Would you like me to send you a copy as well?
Craig E Rasmussen ( 10:47 EST): Have him try the account I set up for him.
peter stauning ( 16:47 GMT+0100): He will !!!
peter stauning ( 16:48 GMT+0100): Hello Ted. Welcome back from the cold Antarctic...peter
peter stauning ( 16:50 GMT+0100): Ted. I'm pleased to hear, that you'll submit a LOI. You ned not send me a copy separately. I'll get that from Eigil

**Figure 5.1: 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.**

Radiologists and doctors collaborate by exchanging such recordings. The sharing, at a later time, of such a workspace is valuable to fellow radiologists (e.g., for intern training), to clinicians (e.g., for consultation and analysis), and to administrators (e.g., as active documentation). Such digital, high resolution session recordings will not only help to capture radiologists' 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.

In this chapter, I examine the notion that applications, their workspaces, and sessions on these workspaces in many cases should be temporally-aware.

## 5.2 Goals and Requirements

Our research is partially motivated by the needs of two projects at the University of Michigan: UARC [18] and the Medical Collab. Specifically, in the Medical Collab, our goal is to support interactive delayed sharing of the workspace of a radiologist. Such workspace consists of tools that allow to: (1) analyze x-rays images, (2) attach audio and graphical annotations, and (3) review textual reports.

The sharing of such workspace is valuable to:

- radiology interns:

  For example, consider the training of radiology interns,

- clinicians:

  For example, consider the support of consultation and analysis, and

- administrators:

  such as RIS/HIS (Radiology and Hospital Information Systems) administrators. For example, consider the use of session objects as a form of, perhaps legally binding, active documentation confirming that the radiologist has performed a service (and thus can be billed now).

From a collaborative viewpoint, our goals *wrt* the sharing of a CSW are to:

- reproduce intra-task content

  that is, at a later time, review the steps taken that led to a diagnosis;

- review the task

  that is, at a later time, browse/annotate such session; and

- reuse the task's results and procedures

  that is, build upon the results contained in the workspace.

A "*replayable workspace*" allows capture, storage, and playback of a session on the collective workspace spanned by multiple applets. Our replayable workspaces are centered around two notions: temporal awareness and replay-awareness. Temporal-awareness refers to the mechanisms (e.g., capture, storage, scheduling, and synchronization) needed to deliver temporal control over inputs to an applet (i.e., applet inputs as a time-dependent media stream). On the other hand, replay-awareness refers to workspace-wide mechanisms needed to enforce homogeneous functional compliance over a collectiveness of temporally-aware applets. While temporal-awareness enables media management, replay-awareness enables workspace management. In this chapter, we explore both these notions.

## 5.3 Applications, Workspaces and Sessions

Our paradigm is centered around object-orientation. Typically, a computer-supported workspace is spanned by multiple object-oriented applications. The user interacts with the workspace applications to perform a task. The creation of a task spans a single user's session with the workspace. Such workspace is represented by three types of objects: (1) application objects, (2) workspace objects, and (3) session objects. Application objects (referred to as applets) are coordinated by the workspace object (referred to as a session manager) in the spanning of a replayable workspace. The representation of a task, (i.e., inputs and state to applications on the workspace), is captured and stored in a workspace-wide distributed artifact (referred to as a session object).

To support later-time orchestration of a workspace, application objects need to provide access to their abstract base machines (as in Parnas [82]). In particular, two types of intra-application interfaces are required: (1) state capture, (2) event capture. To orchestrate application objects on a workspace, the workspace object needs to interface to them. Application objects need to comply with two types of interfaces to the workspace object: (1) inter-application synchronization and (2) functional orchestration. It helps if these interfaces are the same regardless of application objects (i.e., polymorphic). Informally, a replayable application is an application object that complies with the aforementioned interfaces. We have specified the behavior of session objects. Compliance of application objects to these interfaces and the underlying media model associated with our session objects (i.e., temporal media streams) extends a collective of application objects and the workspace spanned by them into our replay-awareness.

## 5.4 Replayable Applications and Workspaces

In this section, I provide insight into the system issues that differentiate replayable applications and replayable workspaces.

### 5.4.1 Replayable Applications

A MacDraw-like object oriented drawing application was retrofitted into replay awareness (see Fig. 5.2. The application allowed tasks such as drawing figures, entering text, moving figures, grouping objects, resizing objects, scribbling, font and color selection, etc..

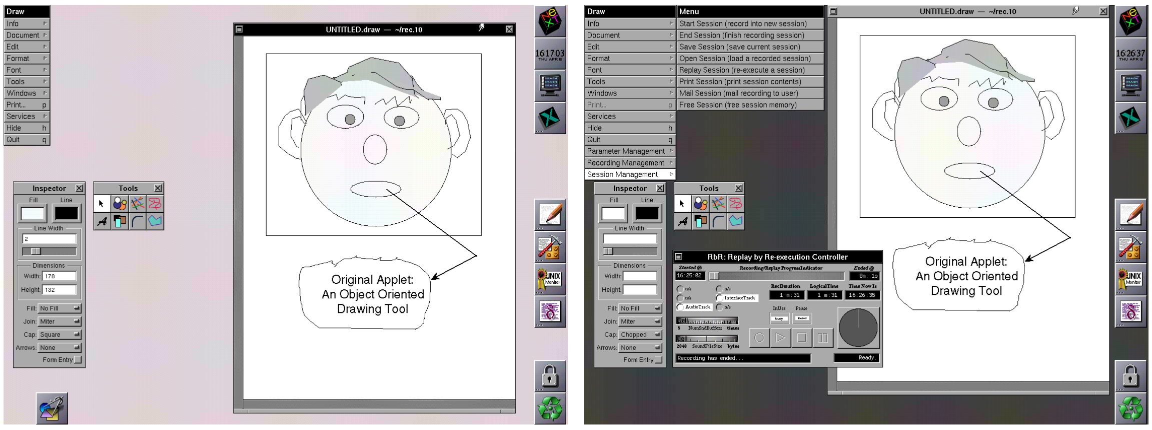

Draw
Info
Document
Edit
Format
Font
Tools
Windows
Print...
Services
Hide
Quit
Parameter Management
Recording Management
Session Management
Menu
Start Session (record into new session)
End Session (finish recording session)
Save Session (save current session)
Open Session (load a recorded session)
Replay Session (re-execute a session)
Print Session (print session contents)
Mail Session (mail recording to user)
Free Session (free session memory)
Inspector
Tools
UNTITLED.draw — ~/rec.10
Original Applet:
An Object Oriented
Drawing Tool
RbR: Replay by Re-execution Controller
Started @
Recording/Replay ProgressIndicator
Ended @
RecDuration
LogicalTime
Time Now Is
AudioTrack
InterfaceTrack
InUse
Pause
NumSndBuffers
SoundFileSize
Recording has ended...
Ready.

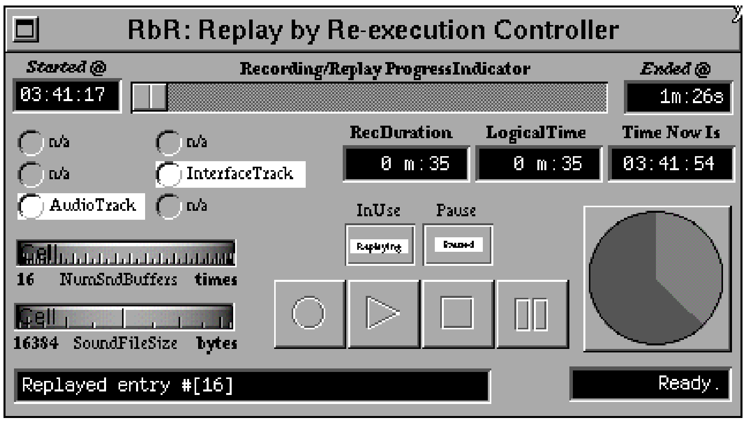
**Figure 5.2: An object-oriented drawing application (left) retrofitted into a replayable application (right, with a dark grey background. Menus and a replayable controller window are added to the application.**

Such monolithic replayable applications access the paradigm features through menus and windows inherited by the application from the replayable object class. For example, the session controller window inherited by a replayable application is shown in Fig. 5.3.

This object-oriented prototype was implemented for NeXT workstations under the Mach OS. The provided the Replayable object class coupled with transparent access to infrastructure services. Such infrastructure allowed the monolithic application to: (1) re-execute window events (e.g., gesturing, typing, moving windows); (2) record audio-annotations; (3) synchronize the playback of these streams; and (4) replay one or both streams.

The infrastructure associated with this prototype provided a *logical time system* (LTS) to support time-stamping of events and frames. It also provides efficient, disk access to streams. Finally, it provides per-stream scheduling and inter-stream synchronization protocols to support faithful playback of streams (see Fig. 5.4).

This prototype supported two streams: the window and the audio stream. Each stream was dispatched to a separate processor. The window event stream was dispatched to the CPU — subject to arbitrary playback conditions. The audio stream was dispatched to the DSP — assumed to be dedicated. These components (application, streams, DSP, CPU, infrastructure services, disk, and data paths) are shown in Fig. 5.5 and 5.6. Fig. 5.5 shows the record-time view. Fig. 5.6 shows the replay-time view of the prototype.



Replayed entry #[16]

Ready.

**Figure 5.3: View of the Replayable application controller. Replayable applications inherit this session controller window. The controller provides session feedback as well as user-level access to operations over session objects such as record, replay, pause, and stream selectivity, among others.**

### 5.4.2 Replayable Workspaces

To explore our replayable workspace ideas, we used the Osiris II tool [59] from the University Hospital of Geneva, to provide the "*look-and-feel*" of the radiology domain. A replayable workspace prototype was constructed as a black box to extend replay-awareness to any X-based application (in this case, an Osiris session), coordinated with the playback of an audio stream and a window display stream. This prototype allowed capture and replay of any such session, annotated with audio. The prototype consisted of an audio tool and an X-windows multiplexer tool retrofitted to be replayable. A session manager was used to coordinate these tools. Snapshots of a sample session with the Osiris application under the Xmx tool are shown in Figs. 5.7-5.9.

## 5.5 Issues for Computer-Supported Workspaces

In this section, we outline fundamental issues on the asynchronous use of CSWs. A CSW is composed of multiple tools. A CSW tool is a user-level process that renders services to the CSW. A service is a user-oriented function provided to the CSW. A tool service (say V) can either be: (1) triggered by a service (say W) (e.g., a causal interaction ($W \leftarrow V$)) or (2) cross-referenced by another (say A). (e.g., a temporal relation ($A * V$) between services).

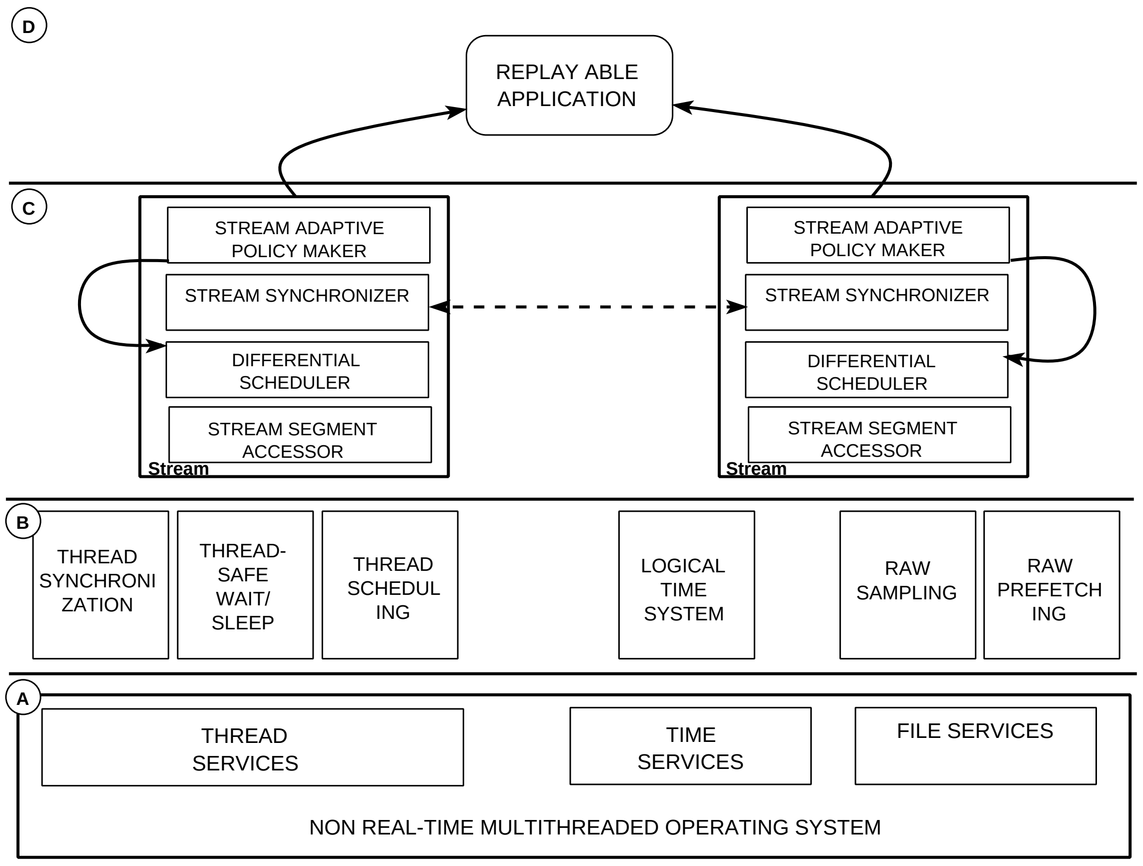


Figure 5.4: The infrastructure of the monolithic replayable application (D). Such replayable application sits on top of modular and independent per-stream handlers (C). Stream handlers are thread-based and rely on shared, fine-tuned middleware services (B) such as logical time systems, mutual exclusion, and synchronization provided by the underlying workstation (A).

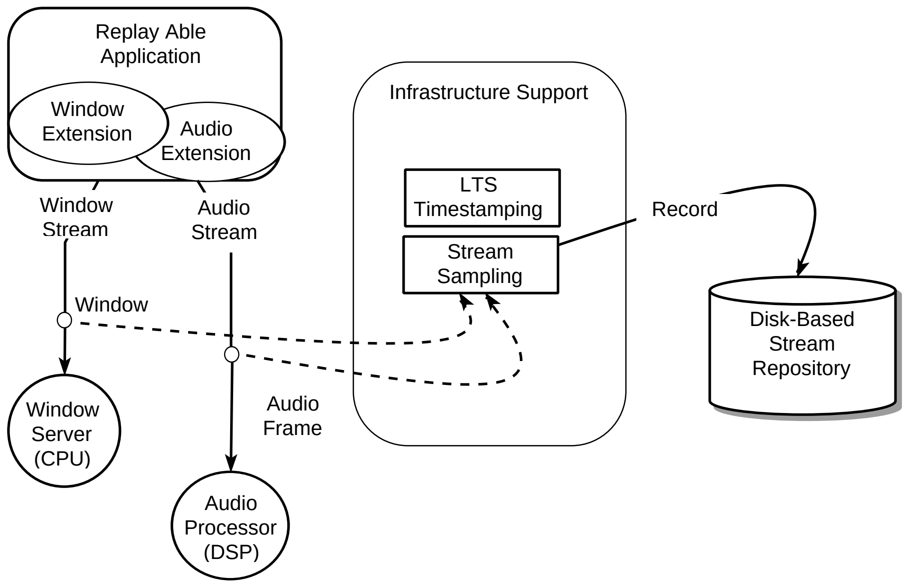


Figure 5.5: Model for record of window and audio streams. The audio and window streams are sampled, timestamped and recorded into a persistent repository.

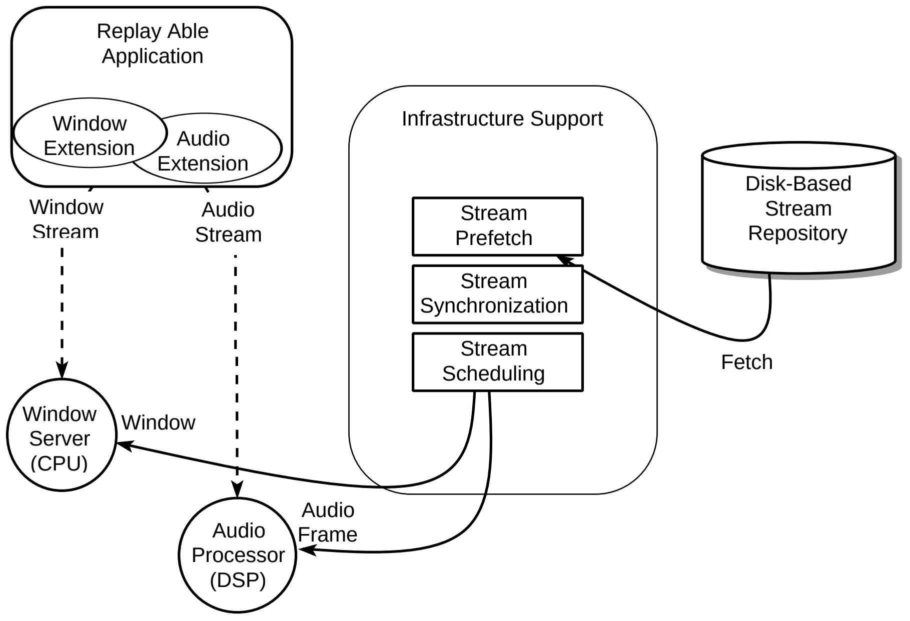


Figure 5.6: Model for replay of window and audio streams. Events are prefetched and dispatched to the corresponding stream of the Replayable application. The infrastructure takes care of maintaining the relative synchronization of the streams and making the necessary adjustments to preserve relative synchronization.

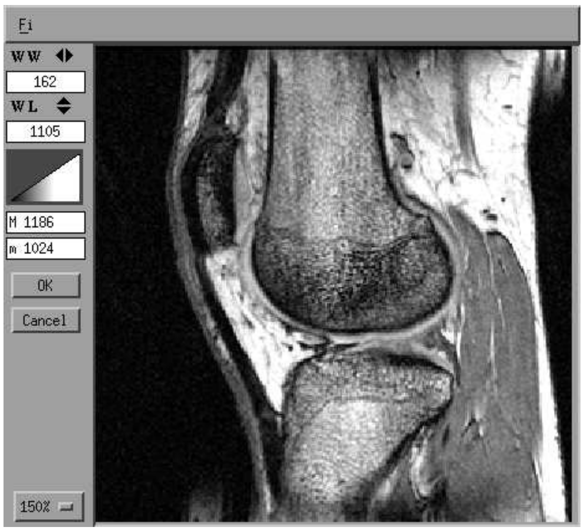


Figure 5.7: Here, an x-ray is loaded into Osiris II, which runs under the X-multiplexer tool.

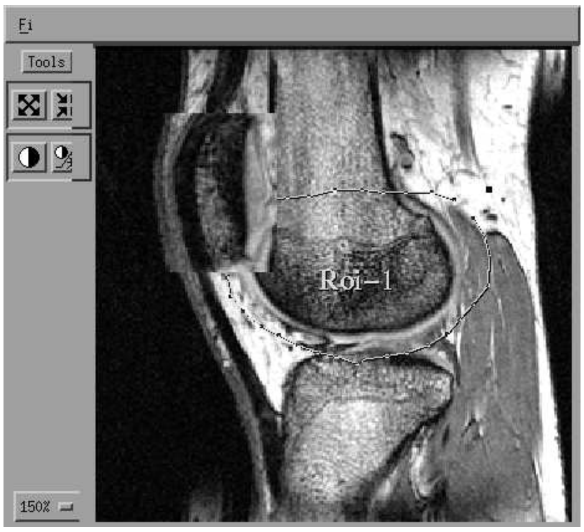


Figure 5.8: Close-up viewing of the x-ray. Using Osiris, a region of interest (Roi-1, middle) is created and a magnifying glass tool is being used to examine details (upper-left of Roi-1).

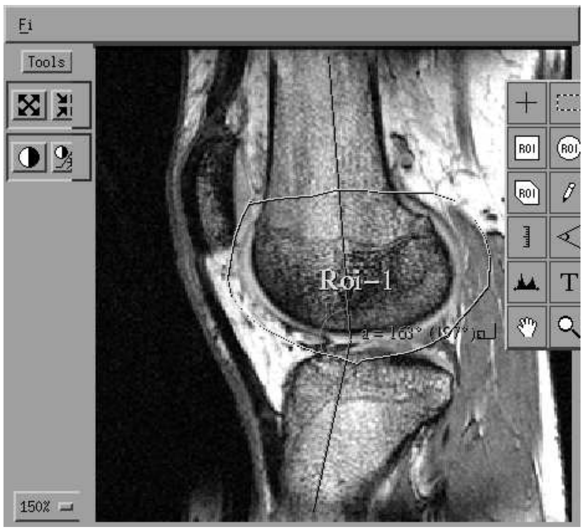


**Figure 5.9: Analyzing the x-ray. A tool palette is accessed (upper right side) and its angle measurement tool is selected. The tool is now being used to measure the angle at which the knee joint is bent (lines on the center). A measurement of 163 degrees can be found (lower right of Roi-1).**

Although each tool may have some degree of temporal-awareness, the composite CSW does not. To support the replay of the workspace spanned by multiple such tools, we must address the following issues:

- tool coordination problem:

  coordinate the services provided by the various tools found in a replayable workspace. For example, it should be feasible to reproduce both causal and concurrent interactions between services of the tools in the workspace.

- media integration problem:

  satisfy temporal constraints during the playback of inputs on these tools. For example, it should be possible to synchronize the asynchronous re-execution of multiple tools and meet playback tolerances to asynchrony and discontinuity of these tools.

### 5.5.1 Modular vs. Monolithic Workspaces

In addressing tool coordination and media integration problems, we must consider how services are to be mapped to tools. There are two basic approaches.

In the first approach, services of multiple tools are integrated into a monolithic-application

— using carefully orchestrated, cooperating threads. Our previous prototype followed this approach [66, 67]. This approach delivers complete control over tool coordination and media integration problems since, at the thread level, fine-grained scheduling and synchronization control is possible. Unfortunately, this low-level control it also limits the flexibility of the resulting infrastructure (see Fig. 5.4).

In the second approach, each tool remains a user-level process in the replayable workspace but it is enhanced with our specification of temporal-awareness. In this approach, the coordination of tool services is deferred to an intermediary process — in our case, referred to as the session manager. This example of tool coordination is typically found in "*publish-and-subscribe*"/"*message-bus*" [22] systems. Although the underlying causal model used by these systems can address our tool coordination problem, it can not address our media integration problem which requires support of fine-grained temporal inter-media relationships. A feasible solution to our tool and media problems should provide causal tool coordination and fine-grained inter-media synchronization. Next, we present an architecture that provides modular support of both tool coordination and media integration.

### 5.5.2 Impact of Media Heterogeneity

To playback a CSW session we need to synchronize the playback of multiple applets. However, because we want to support arbitrary applets, our synchronized playback problem is restricted by the lack of homogeneity in arbitrary applet input streams. Typically, synchronized playback of continuous streams benefits from homogeneous characteristics such as lack of event statefulness, playback continuity requirements, tolerance to playback jitter, etc.. In particular, Tables 5.1 and 5.2 show a comparison of possible applet streams found in typical workspaces. Such applet streams have heterogeneous characteristics: (1) logical data unit (LDU) definition, (2) LDU continuity, (3) LDU statefulness, (4) LDU periodicity, and (5) time variance on LDU re-execution. Applet streams represent re-executable content streams with heterogeneous characteristics with respect to (*wrt*) those of continuous streams such as audio. The integration of re-executable content streams *wrt* continuous media streams is discussed on Chapters 7 and 8.

| Temporal Media Stream | | Degree of Freedom *df* |
|---|---|---|
| Key | Description | example *df* resource |
| $LTS$ | global timing | n/a |
| $A^c$ | continuous audio | n/a |
| $A$ | discrete audio | silences |
| $V$ | video | frame rate |
| $W$ | window stream | inter-event delay |
| $D$ | application data | inter-event delay |
| $C$ | command stream | inter-event delay |

**Table 5.1: Typical streams found in CSWs.**

| LDU Charact. | $A^c$ | $A$ | $V$ | $W$ | $D$ | $C$ |
|---|---|---|---|---|---|---|
| Definition | *frame* | *phrase* | *frame* | *event* | *datum* | *command* |
| Continuity | Y | N | Y | N | N | N |
| Statefulness | N | N | Y/N | Y | Y | Y |
| Periodicity | Y | N | Y | N | Y/N | N |
| Time-Variant | N | N | N/Y | Y | Y/N | Y |

**Table 5.2: Media characteristics of applet stream events.**

## 5.6 Concluding Remarks

There is some experience so far with the paradigm. A team of psychologists performed observational studies on the use of our paradigm in a controlled environment. Their complete findings about the Medical Collab will be reported elsewhere by E. Yakel and T. Finholt. Here we take a moment to review their observations and their implications over our notions of replay-awareness.

- content augmentation:
  The augmentation of a session contents through temporal annotation is essential to its understanding at a later time. However, there are different uses and added value introduced by annotation. In this dissertation, I address the integration of multiple heterogeneous tools for annotation of a replayable workspace. Some of these are the playback of: 1. audio annotations; 2. pointer-based gesturing; and 3. textual annotation tools.

- span of a session:
  Typical sessions span only a few minutes (although this is domain dependent). For example, for the radiology domain, the average duration of a session on the ultrasound domain is about 3.4 *mins* (with a low of 1.1 mins and a high of 7.5 mins). For the less complex, chest domain, sessions lasted an average of 1.56 *mins* (with a low of 0.21 mins and a high of 5.32 mins).

- segment of common interest:
  A general observation across domains is that there are about 45*secs* of interest in a session, in which results of common interest are reached. The authoring of such sessions has three phases: (1) setup, (2) shared result, and (3) wrap-up.

- arbitrary indexing and random access:
  Indexing to a recording is a needed improvement. For example, we would like to allow users to browse the shared-result segment of a session without requiring the playback of the setup segment of a session. The state capture formulation supports indexing into the recordings, however an interface mechanism is needed to allow specification of arbitrary indexes into the recording.

- wrap-up of session:

  Sessions normally end with the same ending sequence on which the recorder reaches for the end of session control button. However to users watching the session, this carries limited content. It wastes user's time. Our observational scientists observed that a hot key to session ending was a necessity.

- workspace-based capture:

  It is essential that record and capture occurs across multiple windows not in a single application. Radiologists and users of computer supported workspaces often perform comparison across elements in their workspaces.

Radiologists in the Medical Collab were exposed to various replay-awareness prototypes. In particular, radiologists were exposed to the paradigm of record and replay through the monolithic replayable drawing application described in Section 5.3.1. In addition, they were also exposed to the replayable workspace prototype described in Section 5.3.2 so as to facilitate the identification of requirements over our paradigm. The following observations were made by radiologists after using these various replay-awareness prototypes.

- The ability to make recordings between arbitrary time periods in an interactive session is very important.
- Being able to go back and edit a recording is important. It is important to radiologists to be able to refine a recording into a polished version.
- Although synchronization of audio and gesturing is important radiologists are used to providing very precise spatial coordinates of regions of interest. As a result, occasional lack of close synchronization is not disruptive whereas audio playback continuity is essential. Furthermore, a coarse grain synchronization (approximately close to that of audio annotated slide shows [98]) appeared satisfactory.
- Availability on heterogeneous platforms is important because of the pervasiveness in hospital environments.

In Chapters 6 and 7, I examine some issues on the support of the first two concerns. In Chapter 8, I describe mechanisms for achieving flexible tradeoffs on media integration. Finally, the issue of platform heterogeneity is addressed through the reliance on workstation-unaware re-execution models such as Java. For this reason, the current prototype of the Medical Collab is implemented in Java.

# CHAPTER 6

# THE ARCHITECTURE OF REPLAYABLE WORKSPACES

*"There is no director ... each is part of the whole and the whole is part of each."*

— *about Javanese ensemble music.*

## 6.1 Introduction

Suppose that while interacting with multiple applications on your desktop, you reach an interesting result. You are likely to document these results and procedures so that maybe, *later on*, you can share these — results and procedures — with your colleagues. Some choose to write scripts detailing the steps and results taken. Others capture these tasks with either analog or digital video. Both approaches are often inadequate for collaborative work due to the potential loss of collaboration content. The approach taken in this dissertation addresses the above problem by supporting the *asynchronous sharing* of an application workspace [67]. This is accomplished by extending an application object with our notions of replay-awareness (i.e., the replayable application object class and its associated session object(s) as described in Chapters 3 and 5).

The rest of this dissertation, however, generalizes the notion of replay-awareness to the support of computer-supported workspaces (herein CSW). A CSW is composed of multiple small tools (referred to as applets) as opposed to one big monolithic application. Such asynchronously-shared CSWs are referred to herein as **replayable workspaces.** Although similar to the term **shareable** workspaces found in synchronous collaborative systems, the term replayable highlights the asynchronous (later-time use) nature of our research. In this chapter, I describe an architecture for the support of replayable workspaces. Its key goals are:

- extensibility:

  That is, the architecture should support workspaces composed of multiple, different applets. It should be possible to record and replay multiple applets, accompanied with temporal annotations such as audio, gesturing, and/or video.

- modularity:

  That is, the architecture should preserve the original applet's awareness (or in particular, unawareness) of other applets in the workspace. It should be possible, when feasible, to *plug-in* a new applet (e.g., an enhanced version of the audio applet) without affecting other applets in the workspace.

- reusability:

  That is, it is desirable, when possible, to reuse existing multimedia applets, such as audio, video, and other applets. Retrofitting an existing applet into a replayable workspace should result in limited and well-contained changes to the applet's implementation.

- collaborative-awareness:

  That is, the architecture should support the collaborative features (such as record, replay, annotation, and editing of session objects) found in the paradigm described in Chapter 3 and in [67].

In this chapter, I analyze the architectural support needed to achieve such flexible and modular replayable workspaces. The rest of this chapter is organized as follows. First, I state the motivation and requirements for my research on replayable workspaces. Second, I examine key issues of the underlying, shared data model, for the support of replayable workspaces. Third, I describe a flexible, media-independent, architecture that addresses the above key goals. Next, I present the interfaces of the APIs of the architecture to allow applets to span a replayable workspace and examine the tool coordination support provided by the architecture. Then, I present my concluding remarks.

## 6.2 Motivation

A "*replayable workspace*" allows capture, storage, and playback of a session on the collective workspace spanned by multiple applets. Our replayable workspaces are centered around two notions: temporal awareness and replay-awareness. Temporal-awareness refers to the mechanisms (e.g., capture, storage, scheduling, and synchronization) needed to deliver temporal control over inputs to an applet (i.e., applet inputs as a time-dependent media stream). On the other hand, replay-awareness refers to workspace-wide mechanisms needed to enforce homogeneous functional compliance over a collectiveness of temporally-aware applets. While temporal-awareness enables media management, replay-awareness enables workspace management. Our motivation is to extend such notion of replay-awareness to computer-supported workspaces.

## 6.3 Requirements

To extend replay-awareness to a workspace, we require flexible mechanisms to: (1) manage heterogeneous tools on a replayable workspace (i.e., applets); (2) model the relationships between these applets; (3) characterize the integration of the heterogeneous streams supported by these applets; and (4) provide mechanisms for playback integration of these heterogeneeous streams; as outlined below.

- management of heterogeneous workspaces:

  We desire mechanisms for workspace management to be independent of the applets that compose the replayable workspace. This is the subject of this chapter, i.e., Chapter 6 (see also [70]).

- temporal modeling of heterogeneous streams:

  We desire mechanisms to characterize and enforce heterogeneous relationships between re-executable content streams and other streams such as continuous streams of applets on a workspace. This is the subject of Chapter 7.

- integrated playback of heterogeneous streams:

  We desire to formulate a playback media integration model that removes the heterogeneity introduced by re-executable content streams and it is yet flexible enough to incorporate new and *unknown* stream types to be found in future replayable workspaces. This is the subject of Chapter 8.

- performance characterization for heterogeneous streams:
  Synchronization and continuity are competing processes. Intuitively, it is possible to achieve good playback continuity and relatively poor synchronization as it is possible too, to achieve poor playback continuity with relatively good synchronization. While is is easy to relate continuity metrics to continuous media, it is not for re-executable streams. For re-executable streams, I propose to relate it to the smoothness of the execution of asynchronous events. Intuitively, we desire to avoid abrupt updates to the applet as these have the same effect as playback discontinuity for continuous streams. This is discussed on Chapter 8.

This chapter focuses on middleware and its associated framework for extending and managing replay-awareness services on the workspace spanned by multiple applets. As also reported in [7], the notions of tools, API transparency, platform/tool heterogeneity, and a shared data model are essential characteristics to described of middleware services. I examine these notions below (also described in [69]).

## 6.4 Overview of the Architecture

A CSW is composed of multiple tools. A tool is incorporated into a replayable workspace by means of a user-level process, referred to as a *stream controller*. A stream controller extends *temporal-awareness* to a particular tool service and its associated temporal media stream. A user-level process, referred to as a session manager, coordinates these temporally-aware tools (stream controllers). The session manager interfaces to and manipulates these stream controllers through simple "*VCR-like*" commands. These polymorphic commands are both tool and media independent, (i.e., are the same regardless of the services provided by individual tools). Fig. 6.1 illustrates the architecture of our replayable workspace [69].

The stream controller abstraction has three components:

- temporal controller
  the glue logic to our replayable workspaces. Delivers temporal-awareness to its CSW

tool.

- temporal media stream
  herein, applet stream. The applet stream represents a temporal ordering of the inputs to a tool and its services.

- re-execution device
  the underlying media device or the tool itself. The re-execution device is modeled as a consumer/producer of applet stream events.

### 6.4.1 Retroffiting Impact

The delivery of my collaboration mechanisms (i.e., replay-awareness and temporal-awareness) to an object-oriented applet is accomplished through the stream controller abstraction. The stream controller abstraction is a new object pattern used to deliver temporal mechanisms (e.g., scheduling, synchronization, etc.) with reduced retrofitting impact over an object-oriented applet. The stream controller abstraction model has three components: (1) the temporal controller, (2) the applet input stream, and (3) the applet itself. To the developer of an applet, the temporal controller represents the heart of our replayable workspace mechanisms.

The stream controller abstraction provides an applet with the necessary interfaces for the integration of an applet into our replayable workspaces. These interfaces are:

- workspace interfaces:
  The replayable workspace architecture requires that applets be able to "plug-in" to our server, the session manager, for workspace management. The temporal controller hides the complexity of such interfaces from applet developers. Effectively, applets are unaware of our server-based orchestration of their collective workspace.

- functional compliance:
  The management of applets in a workspace requires individual applets to comply with the application programmer interfaces (APIs) to our replayable workspaces. These APIs are described later, in the Section 6.6. The applet controller provides applet developers with default callbacks to our replayable workspace APIs. These callbacks act as placeholders for applet-specific handling of replayable workspace commands

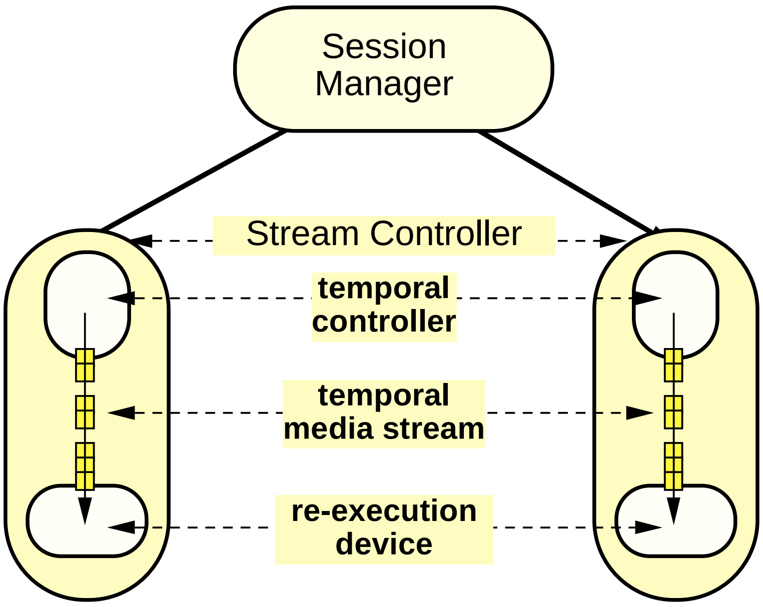


Figure 6.1: The architecture of replayable workspaces. A replayable workspace is composed of several replayable applets. The stream controller abstraction is used to share a common understanding of temporal-awareness and its mechanisms to applets in a workspace. The abstraction has three components: (1) the temporal controller, which encapsulates mechanisms for temporal-awareness as well as interfaces for workspace replay-awareness; (2) the applet stream, which represents a temporally-ordered sequence of inputs and state to the abstract base machine of an applet; and (3) the applet itself, which provides a deterministic abstract base machine for the re-execution of inputs and state. Stream controllers are coordinated by an intermediary process referred to as a session manager. The session manager delivers replay-awareness to the workspace and provides a unified view to such a replayable workspace.

such as record(), replay(), save(), pause(), forward(), etc.. The applet developer must override these placeholders with applet-specific processing that complies with the semantics of these polymorphic callbacks.

- temporal-awareness:

  The APIs to our replayable workspaces are media-independent. All media processing occurs outside the session manager server. The temporal controller eases the complexity of media processing from applet developers by providing them with (1) a common, media-independent, data model for the representation of temporal media streams. (2) a common set of primitives to handle events from a temporal media stream (i.e., capture(), playback(), store(), fetch(), etc.) events from an applet's input media stream.

The common data model is specified next. The event handling primitives are formally specified in Chapter 9.

## 6.5 Shared Data Model

A flexible workspace requires decoupling the logical representation of a session object from the physical representation of any of its applet streams. Informally, the common data model is specified as follows: (1) a session is composed of one or more streams; (2) streams are composed of one or more sequences; (3) sequences are composed of one or more events; and (4) events are applet-specific.

A two level data model is used for the representation of session objects. The intra-tool data model (or media model) refers to the physical layout of an applet stream. The media model is used to optimize physical access to stream events (e.g., prefetching, random access, storage, etc.) — see Fig. 3.3. This needs to, and thus is only known, to its stream controller. The inter-tool data model (or workspace model) refers to the logical representation of a workspace session. Both the workspace and media model need to be agreed to by all streams controllers so as to enforce polymorphic manipulations over a session. These data models are formally specified in Appendix A.

The media model is represented through a temporal media stream. This temporal media stream, referred to as an applet stream, models operations over an applet as a temporally-ordered sequence. An applet stream is composed of events and state(s). Events represent units of re-execution; as such they are defined by their respective applet (i.e., the underlying re-execution device). Events represent deltas applied over the state of an applet. State data provides the basis for a restartable process formulation that allows re-execution of events at an arbitrary time-index.

The media model provides media decoupling of the session manager from its respective stream controller. The session manager remains unaware of stream-dependent details such as

- event granularity:
  the definition and granularity of applet stream events is known only within an applet temporal controller. Transparency of event granularity allows the use of abstract commands as stream events. However, the selection of grain for stream events also bounds the grain (and efficiency) of synchronization mechanisms. Granularity transparency is important for the construction of polymorphic workspace operations.

- naming scheme:
  the naming scheme used for applet stream events is known only within an applet temporal controller. Naming transparency is important for the specification of polymorphic workspace operations.

- data location:
  the location of events used by applet stream events is known only within an applet temporal controller. However, location transparency assumes media access operations (such as prefetch(), store()) can be implemented efficiently for the event granularity; (as otherwise this would affect the efficiency of scheduling mechanisms). Location transparency is important for the specification of transportable workspace objects.

- data access model:
  the access model used for applet stream events is known only within an applet temporal controller. For example, the temporal controller of an applet may support random access to an event through either of the following modes:

  - real random access:
    For example, consider random access over the audio stream by restarting play-

back at the actual byte that corresponds to the requested event.

- approximated random access:

  For example, consider approximated random access over an MPEG video stream by jumping to the closest *I-frame* coupled with sequential playback of *P- or B-frames.*

- simulated random access:

  For example, consider random access over a re-executable content stream through sequential re-execution of its events.

Access transparency is important for the construction of polymorphic workspace operations.

- re-execution device:

  the actual device used for the re-execution of applet stream events is known only within an applet temporal controller. The "*devices*" may again be a real device such as an audio processor or a simulated device such as an applet. Device transparency is important for the specification of transportable workspace operations.

- data layout:

  the layout scheme used for applet stream is known only within an applet. For example, an applet stream can be implemented as a temporal log or a collection of temporally ordered files or shared objects as needed by the implementation of an applet temporal controller. Media layout transparency is important for the specification of portable workspace operations.

The workspace model imposes session and workspace semantics over the media model of multiple applets on a workspace. The workspace model views the workspace as composed of abstract streams. Streams on a session object are rooted as a file hierarchy of media directories [67]. The data of a stream controller is represented using standard Unix filesystem support (i.e., byte-stream files and directories). The session object is thus copied and transported as any other directory.

The workspace model addresses the representation of a workspace. Session objects are transportable objects (as the authored sessions from CMIF [14]). During playback, the relative temporal progress between its abstract streams must be preserved. To support this, the workspace model characterizes the playback (i.e., scheduling and synchronization) of multiple abstract streams in terms of: (1) the relative temporal progress characterization (relationships) that describe the concurrent playback of multiple abstract streams; (2) the tolerance asynchrony on such relationships; (3) the degrees of freedom found on such relationships for the adjustment of scheduling; (4) and the potential treatments for correct scheduling and synchronization departures on these relationships. Workspace modeling is formally analyzed in Chapter 7.

## 6.6 The Replayable Workspace

A computer-supported workspace is composed of multiple applets. An applet is incorporated into the replayable workspace by means of a stream controller. The stream controller abstraction is used to share a common understanding of temporal-awareness and its mechanisms to applets in a workspace. The stream controller abstraction has three components: (1) the temporal controller, which encapsulates mechanisms for temporal-awareness as well as interfaces for workspace replay-awareness; (2) the applet stream, which represents a temporally-ordered sequence of inputs and state to the abstract base machine of an applet; and (3) the applet itself, which provides a deterministic abstract base machine for the re-execution of inputs and state. A temporally-aware applet (i.e., a stream controller) is referred to as a replayable applet.

A replayable workspace is composed of several replayable applets. Replayable applets are coordinated by an intermediary process referred to as a session manager. The session manager delivers replay-awareness to the composite workspace spanned by these applets and provides a unified user interface to the resulting replayable workspace. Fig. 6.1 illustrates the architecture of our replayable workspaces.

### 6.6.1 Stream Controllers and Temporal Awareness

A stream controller delivers temporal awareness to its applet through two abstractions: media access and media control.

Media access provides low-level device primitives for media I/O processing. Two such primitives provide event-based I/O handling:

- event = read(index)

This primitive reads the requested index on an applet stream and returns the associated event.

- index = write(event)

  This primitive sequentially writes an event on an applet stream and returns its associated index.

These event-based I/O primitives are coupled with block-based I/O primitives. Ideally, a block should be composed of one or more event sequences. Block-based I/O primitives retrieve data into per-stream event caches. On the other hand, event-based I/O primitives retrieve from the event caches or when a miss occurs, from the stream data file(s). The block-based I/O primitives are:

- event = prefetch(index)

  This primitive prefetches and caches a continuous block of events *wrt* the requested index.

- index = buffered-write(event)

  This primitive buffers contiguous events and writes then as a block.

The implementation of these primitives is discussed in Chapter 9.

Media control provides: (1) intelligence to invoke media access primitives in such a way so as to extend temporal-awareness to the re-execution device and (2) interfaces to sequence-oriented operations such as adaptive scheduling, performance monitoring, and prefetching. Temporal-awareness associates a temporal media stream to the record and replay of the abstract base machine of an applet. Sequence-oriented operations are included in the generic temporal controller specification. The implementation of the temporal awareness and sequence-oriented management is discussed in Chapters 3 and 7.

### 6.6.2 The Session Manager

The session manager is a user-level process that coordinates temporally-aware tool services from the various stream controllers. The session manager allows a user to manipulate a session on a replayable workspace through simple *VCR-like commands* — regardless of the tools and the media found on the workspace. Such polymorphism requires the session manager to be decoupled of:

- different characteristics between tool services:

  as such would limit the flexibility and generality of the tool coordination mechanisms.

- different characteristics between applet streams:

  as such would limit the flexibility and generality of the media integration mechanisms.

To decouple the session manager from heterogeneity across applets, homogeneous, polymorphic, interfaces are specified between the session manager and its applets. Stream controllers enforce a homogeneous interface to media-dependent tool services. For example, a request to "*playback a session*" gets broadcast to stream controllers as the abstract command: "*replay stream*".

To decouple the session manager from heterogeneous media handled by applets on its workspace, its interfaces are specified over a media-independent stream type. Media-independent commands are dispatched from the session manager to stream controllers. Stream controllers provide media-dependent execution of commands. These commands are handled by the applet's temporal controller; which provides, for example, functionality such as: (1) record, representation, and re-execution; (2) scheduling and synchronization; and (3) measurements collection. The implementation of media-dependent functions is thus transparent to the session manager.

### 6.6.3 Inter-tool Interfaces

To support tool coordination and media integration, every stream controller must comply with three inter-process interfaces *wrt* the session manager (Fig. 6.2):

- functional interface (F):

  supports tool coordination — its primitives provide abstract machines that build the features of session objects.

- synchronization interface (S,D):

  supports media integration — its primitives support fine-grained inter-media relationships between applet streams.

- feedback interface (M):

  supports the evaluation of the efficiency of the above interfaces.

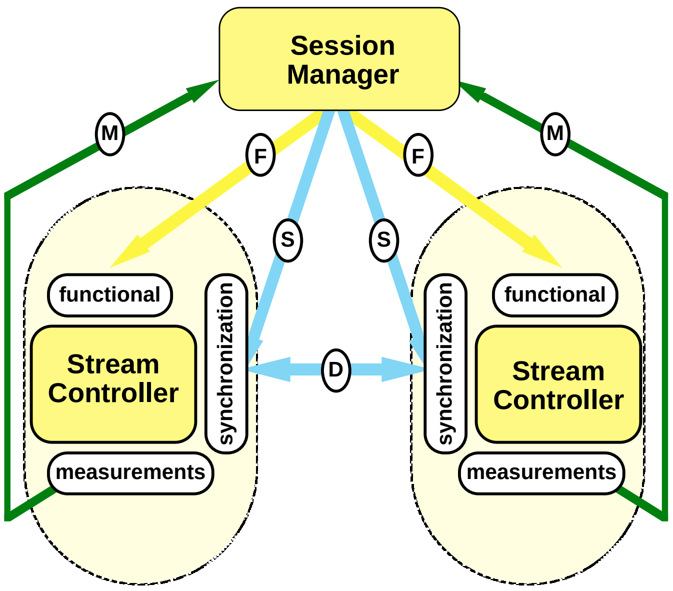


Figure 6.2: Interfaces of the architecture. The architecture uses a client – server model. The session manager coordinates multiple CSW tools — each managed by a stream controller. A stream controller facilitates compliance of an applet to the interfaces of our replayable workspaces: (1) functional interface $F$ (a homogeneous interface to tool coordination), (2) synchronization interface $S, D$ (primitives for media integration), and (3) feedback interface $M$ (efficiency measurements evaluation primitives).

When a stream controller complies with our interface specifications, its corresponding applet is said to be **replayable**. A replayable applet represents a temporally-aware plug-in component to a replayable workspaces. Among other things, it is able to record and re-execute the tool services of its replayable applet.

### 6.6.4 Transport Layer

To preserve the openness and scalability of the architecture, we also specify the inter-process communication (IPC) paths between session manager and stream controllers. A forward-channel $F(str)$ links session manager to a stream controller $str$ whereas a back-channel $B(str)$ links $str$ to its session manager. Since our interfaces are mapped to these IPC paths, channels need to be at least reliable FIFO point-to-point links. By decoupling interfaces from message delivery, we remove assumptions that a transport layer may impose — for example, we make no assumption about delivery mechanisms such as multicasting between session manager and stream controllers.

## 6.7 Replay-Awareness Compliance

In this section, we specify the interfaces that support tool coordination and media integration. Through compliance with these interfaces, a stream controller delivers temporal-awareness to tool services on its respective applet as well as a new plug-in component to the replayable workspace.

### 6.7.1 The Functional API

These primitives support tool coordination. The session manager translates user requests into abstract commands, forwarded to stream controllers. These media-independent commands enforce a homogeneous interface to media-dependent services of heterogeneous stream controllers. These commands use the forward-channels $F()$.

SELECT($str, enable$): *enables* or *disables* stream controllers until further notification. The parameter *enable* indicates whether or not $str$ is to be enabled.

RECORD($str$): a *nonblocking* call that enables stream capture at a receiving stream controller $str$.

END-OF-RECORD($str$): disables $str$ stream capture.

SAVE($str, basename$): causes $str$ to create persistent representation(s) of its temporal media stream. The parameter *basename* contains the path name to the storage hierarchy of the session object.

OPEN($str, basename$): causes $str$ to load the persistent representations of its stream into memory. The parameter *basename* contains the path name to the storage hierarchy of the session object being loaded.

REPLAY($str, s_i$): enables $str$ to chain-replay from the current scheduling interval up to the $i^{th}$ scheduling interval, inclusive.

SET-SPEED($str, speed$): requests $str$ an unconditional and absolute change of its LTS scheduling rate to the value specified by the parameter *speed*.

$s_j$ = PAUSE($str$): requests $str$ to pause its processing at the end of its next feasible scheduling interval (say $s_j$). The scheduling interval $s_j$ is returned to the session manager.

RESUME($str, s_j$): causes $str$ to resume processing but only up to its $s_j$ scheduling interval.

BROWSE($str, s_i$): causes $str$ to launch a content browser of its temporal media. The content browser uses the time position implied by $s_i$ to display the static contents of $str$ temporal media stream.

ABORT($str$): causes $str$ to abort its current operation.

### 6.7.2 The Feedback/Measurements API

These feedback primitives allow the session manager to evaluate the efficiency of media integration and tool coordination. These messages use the back-channels $B()$.

DID-INIT($str$): reports that $str$ has initialized.

CMD-COMPLETED($str$): reports *explicitly* that $str$ has completed processing of its last command.

SYNC-ISSUE($lhs, s_i$): requests (during capture of a session) a synchronization relationship between this $lhs$ to its $rhs$ stream controllers to be established at the synchronization event $s_i$.

SYNC-UPDATE($str, i + 1, start_i, end_i, \omega_i$): triggered by the scheduling of a synchronization event $s_{i+1}$, causes $str$ to collect scheduling measurements $(start_i, end_i)$. The times $start_i$ and $end_i$ represent CSW-wide timestamps for the start and end, respectively, of the processing of the *previous* synchronization event $(s_i)$ at $str$. The value $\omega_i$ is used by adaptive protocols and represents the compensation factor currently in use by $str$'s LTS. For non-adaptive protocols, it is set to 1.

### 6.7.3 The Synchronization API

These primitives provide support needed by a variety of synchronization treatments and protocols.

SYNC-INSERT($lhs, rhs, s_i$): inserts a synchronization event $s_i$ into $rhs$ *wrt* an $lhs$ stream controller — thus establishing a relative synchronization relationship at $s_i$, i.e., $s_i(lhs \leftarrow rhs)$.

SYNC-CONT($str, s_i, \epsilon, \Delta$): terminates the synchronization handshake with $str$ (i.e., releases synchronization block and resumes scheduling). The parameter $\epsilon$ specifies an estimate of the inter-stream asynchrony between $rhs$ and $lhs$ stream controllers. The parameter $\Delta$ is used by adaptive protocols and specifies a recommended adaptation effort. For non-adaptive protocols, it is set to 0. A positive $\Delta$ suggests a rate increase whereas a negative $\Delta$ suggests a decrease.

## 6.8 Flexible Tool Coordination

Next, we show how the tool coordination abstract machine supports the specification of the features of session objects. For brevity, we focus only on browsing features. Temporal access controls (TAC) allow the modeling of "VCR-like" (browsing) features as temporal transformations over a logical time system (LTS), as for example found in [2, 62, 85, 89]. Building on that model, I then show that the browsing support of the architecture are independent of both tool and media characteristics. Throughout the following discussions:

(1) recall that a session object is composed of scheduling intervals, bounded by scheduling events $(s_1, \cdots, s_n)$ and (2) let $str$ be the set of all stream controllers in a replayable workspace.

### 6.8.1 Replaying

The Replay() command enables the playback of a block of scheduling intervals. The following enables chained playback from the current scheduling interval (say $s_k$) to $s_n$: $\texttt{REPLAY}(str, s_n)$.

Internally, stream controllers implement this command as a "back-to-back" chaining of scheduling intervals. The Replay() command only *enables* playback, whereas the actual dispatching of scheduling intervals is controlled by the media integration mechanism. The media integration mechanism executes between scheduling intervals. A naive chaining implementation is likely to introduce intra-media discontinuities (such as media playback continuity glitches). However, our architecture empowers stream controllers to stream-dependent optimizations to compensate for this. For example, based on the strong sequential-access pattern of this paradigm, prefetching can be used to ameliorate this effect.

Finally, scheduling intervals also allow the session manager to support user interactivity during the replay of a sesion. For example, before the dispatch of the next scheduling interval, the session manager can (1) perform housekeeping (such as updating progress feedback) as well as (2) interact with the user (for example, pause the session).

### 6.8.2 Pausing and Resuming A Replay

A Pause() command requests stream controllers to stop playback at the end of the next feasible scheduling interval. However, stream controllers may stop at different scheduling intervals (say $s_j^i$). Consequently, the session manager determines a common scheduling interval $s_j$ (for example, $s_j = max_i(s_j^i)$) across its stream controllers. The session manager then requests stream controllers to playback up to the common scheduling interval $s_j$. The following models such interaction (pause/continue) over the playback of a session object:

$$\left\{ \begin{array}{l} \texttt{REPLAY}(str, s_n); \\ s_j = \texttt{PAUSE}(str); \texttt{RESUME}(str, s_j); \\ \texttt{REPLAY}(str, s_n); \end{array} \right\}$$

### 6.8.3 Browsing

Several forms of browsing are of interest to us.

- Dynamic browsing. To fast-replay a session, the session manager requests its stream controllers to increase their presentation rate. For example, fast-forward from $s_i$ to $s_j$ is modeled as:

$$\left\{\begin{array}{l} s_i = \texttt{PAUSE}(str); \texttt{RESUME}(str, s_i); \\ \texttt{SET} - \texttt{SPEED}(str, speed); \\ \texttt{REPLAY}(str, s_j); \end{array}\right\}$$

- Static browsing. A stream controller is responsible for providing static browsing of its applet stream. For example, an audio tool can display waveforms whereas an application can display a temporal ordering of its data inputs. Browsing at $s_i$ is modeled as:

$$\left\{\begin{array}{l} s_i = \texttt{PAUSE}(str); \texttt{RESUME}(str, s_i); \\ \texttt{BROWSE}(str, s_i); \end{array}\right\}$$

- Random access to an arbitrary scheduling interval $s_i$. Session objects are (very) stateful objects. If state capture is feasible for a stream controller (e.g., as with object-oriented, open-interface classes like in Java), we can rely on periodical state snapshots so as to jump to the nearest snapshot before $s_i$ and fast replay from there on. Alternatively, if state capture is infeasible, we can rely solely on fast replay. The $\textsf{Forward}()$ command encapsulates such media awareness:

$$\left\{\begin{array}{l} \texttt{FORWARD}(str, s_i; speed); \\ s_i = \texttt{PAUSE}(str); \texttt{RESUME}(str, s_i); \end{array}\right\}$$

### 6.8.4 Dubbing

Dubbing of a applet stream is a complex operation because of: (1) intra-media statefulness and (2) inter-media synchronization. To preserve statefulness, dubbing is restricted to well-defined boundaries only (i.e., state snapshots). To preserve synchronization, dubbing is restricted to *rhs* streams. To dub an *rhs* stream, the session manager requests: (1) the

$rhs$ stream controller to record while (2) the $(str - rhs)$ other stream controllers replay. Synchronization events are inserted into the $rhs$ stream by the session manager on behalf of $lhs$ (by means of the Sync-Insert() command. The following dubs $s_{i+1}$ through $s_{j-1}$:

$$\left\{ \begin{array}{l} s_i = \texttt{PAUSE}(str); \texttt{RESUME}(str, s_i); \\ \texttt{RECORD}(rhs); \\ \texttt{REPLAY}(str - rhs, s_{i+1}); \\ \texttt{SYNC} - \texttt{INSERT}(rhs, s_{i+1}); \\ \ldots \\ \texttt{REPLAY}(str - rhs, s_{j-1}); \\ \texttt{SYNC} - \texttt{INSERT}(rhs, s_{j-1}); \\ \texttt{END} - \texttt{OF} - \texttt{RECORD}(rhs); \\ s_j = \texttt{PAUSE}(str - rhs); \texttt{RESUME}(str - rhs); \end{array} \right\}$$

## 6.9 Media and Workspace Coupling/Decoupling

The architecture provides a view of the replayable workspace that allows the designer of a replayable workspace to be unaware of the underlying applets and media found on the workspace. Transparency of media integration is essential to simplify the modeling of workspace features. Such transparency is achieved through "*behind the curtains*" media integration checks at the session manager. To reduce processing overheads, media integration is performed at intervals. However, *unlike in the integration of continuous media*, the interval between integration checks for heterogeneous streams is not either constant nor predictable — since the playback of events is asynchronous and consequently, lacks predictable playout time. The media coupling/decoupling problem is the subject of Chapter 8.

## 6.10 Concluding Remarks

In this chapter, I described a flexible architecture for replayable workspaces spanned by a collective of heterogeneous tools. I presented architectural arguments for the delivery of flexible mechanisms that address the heterogeneity of tools and media on a workspace. In doing so, I introduced the notions of temporal-awareness and replay-awareness so as to deliver a common understanding of temporal orchestration to heterogeneous tool and media in a workspace.

The architecture is flexible in the sense that it removes heterogeneity from the tools and their underlying temporal media streams. The architecure allows developers to remain unaware of differences between the applets that compose a replayable workspace. In particular, the architecture provides an flexible abstract base machine for the specification and modeling of workspace features and services. In this chapter, I showed the specification of services such as interactive browsing. I showed the specification of such features and services to be unaware of the underlying media processing (i.e., media-independent).

Having presented an architecture for the support of replayable workspaces, in the next chapter, I present the specification and modeling of such workspaces.

# CHAPTER 7

# SPECIFICATION AND MODELING OF REPLAYABLE WORKSPACES

*"I never think of the future. It comes soon enough."*

— *Albert Einstein.*

## 7.1 Introduction

A computer-based workspace consists of multiple tools or applets. To illustrate this, consider a workspace $W$ composed of four applets (tools) $(A, B, C, D)$ as shown in Fig. 7.5, where: (A) is a notebook tool e.g., an electronic notebook); (B) is a graphing tool (e.g., a radar stream viewer); (C) is a (shared) annotation tool (e.g., audio annotation tool); and (D) is a (shared) gesturing tool (e.g., workspace-wide pointer gesturing). Each tool manipulates a temporal media stream consisting of a time-ordered sequence of inputs and state. For example, events in the applet stream of A consist of operations such as Insert() and Delete() of editor context, events on B's applet stream consist of operations such as Update() and Refresh() of time series data, similarly, events in C consist of operations such as Play() and Record() of frames of audio data, and events in D consist of operations such as Get() and Dispatch() of window events. During the capture of a session, these inputs are time-stamped and recorded. The playback of such session requires temporal orchestration of the re-execution inputs on these tools or applets.

Applet input streams (herein applet streams) contain re-executable events such as simple window events or even abstract operations. Because the re-execution of these events is

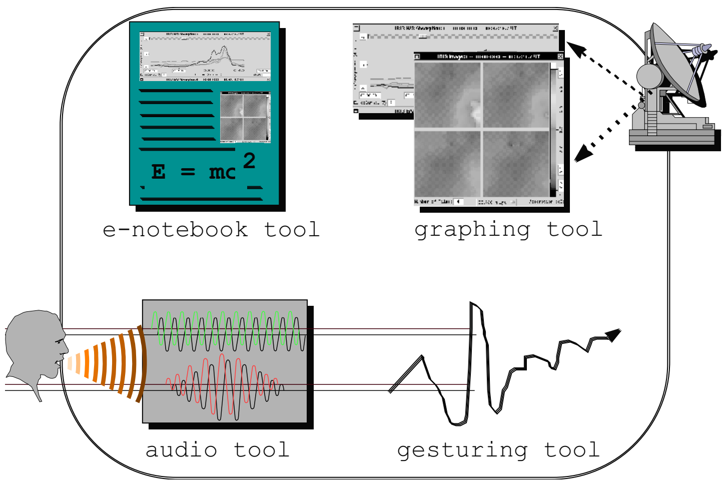


**Figure 7.1: A workspace $W$ consists of multiple tools or applets. The above workspace is composed of four tools: (A) an electronic notebook tool, (B) a graphing tool, (C) a audio-annotation tool, and (D) a pointer-gesturing tool. Some tools provide services shared by other other tools in the workspace. For example, tools (C) and (D) provide shared services. During the capture of a session, inputs to these tools are time-stamped and recorded. The playback of such session requires temporal orchestration of the individual execution of these tools.**

asynchronous and stateful, playback mechanisms need to account for performance differences between record/replay workstation conditions. Furthermore, playback is complicated by the need to preserve different types of temporal relationships. For example, the playback of the notebook and graphing tools requires enforcing a causal relationship (such as for cut and paste) between them. On the other hand, the playback of audio-annotated pointer gesturing requires a purely temporal (non-causal) relationship (see Fig. 7.5).

Relationships among events across applet streams may be different. By taking advantage of (1) the tolerance of a relationship to asynchrony and (2) the tolerance of stream scheduling to discontinuities, it is possible to enforce multiple such relationships during playback. The heterogeneity of these streams and their inter-relationships affect the efficiency of streaming mechanisms. In this chapter, I present a framework for the integration of heterogeneous relationships between $n$ applet streams. The framework and its mechanisms are tool and media independent.

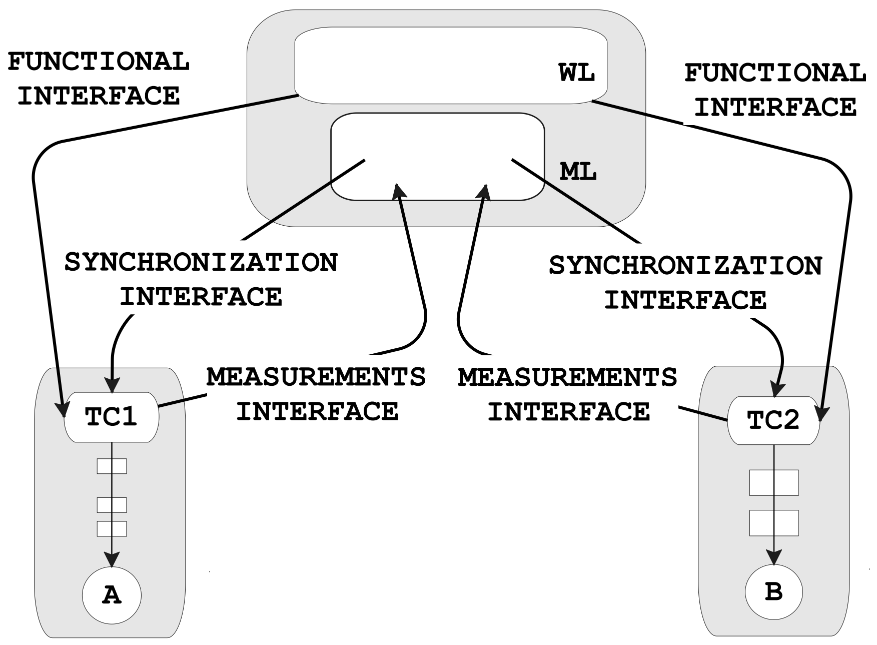


**Figure 7.2: When applets comply with our replay-awareness interfaces, our session manager is able to orchestrate their playback. The session manager uses the synchronization interface to enforce media-independent relationships between multiple applet streams. Applets provide periodical feedback reports about the efficiency of session management via the measurements interface.**

## 7.2 Motivation and Requirements

Our motivation is to extend replay-awareness to computer-based workspaces. The requirements for the support of replayable workspaces were presented in Chapter 6. In this chapter, I examine the specification and modeling of replayable workspaces so as to support the integration of heterogeneous streams. We desire to formulate a temporal specification model that can capture the heterogeneity of relationships between tools in a workspace while preserving the media-independence constraint (see Chapter 6) on which the architecture is founded on.

## 7.3 Specification and Modeling

Tool and media independence are central to our framework as mechanisms for the removal of heterogeneity between applets. A replayable applet is a small application, tool, applet, or device driver, that has been expanded to incorporate these mechanisms. A replayable applet is as unaware of other applets as the original applet was. Replayable applets are coordinated by a centralized process, referred to as the session manager. To replayable applets, media processing occurs *"behind the curtains"*. This applet-transparent media processing takes place at the session manager. The session manager also provides a unified user interface to the workspace spanned by its applets. Workspace management is enforced by the session manager, through operations such as $\{\ f_i =$ record(), replay(), goto(), edit(), pause(), continue(), $\cdots\ \}$ that operate over one or more applets. The developer of a replayable applet is responsible for the implementation of these operations. Media processing is approached abstractly; the session manager is only aware of media-independent information such as (1) the number of applets on its workspace, (2) their inter-relationships, and (3) the tolerance of these relationships. This information is used in the characterization of the playback requirements of a workspace.

### Decoupling of Tool and Media

Tool and media are not totally decoupled. Workspace management is coupled to media processing as follows. Before initiating a new workspace management operation $f_i(...)$; the session manager waits for the stabilization of pending media processing, i.e., $f_{i-1}(), \cdots$.

### Workspace

A computer-based workspace $W$ is spanned by a set of tools or applets that provide integrated services to allow one or more users to perform a domain-specific task. This sort of workspace arises in domains such as space-science and computer-supported radiography. Informally, a workspace $W$ is described by three tuples: (1) $W_A$: the applets found in the workspace; (2) $W_R$: the relationships between these applets; and (3) $W_T$: the tolerances of these relationships.

### Applets

In our context, an applet $A_i$ is a small and simple application such as a component-based tool, an active-pad control, a java applet, or even a device driver. Typically, applets have a single-purpose, well-defined, abstract base machine.

### Abstract Base Machine

The notion of an abstract base machine was defined by Parnas [82]. Let $abm(A_i)$ be the abstract base machine of applet $A_i$, where $abm(A_i)$ is composed of abstract operations $(f_1(), f_2(), \cdots, f_n())$.

### Tool Services

Tool provide integrated services that spanned the workspace. A tool renders a user-oriented, well-defined, service to the later-time reuse of a workspace. Example of tool services are the record/replay of audio and pointer gesturing. A well-defined tool service $TS(A_i)$ is just a user-level representation of an underlying, well-defined, abstract base machine, i.e., $TS(A_i) = f(abm(A_i))$.

### Relation Between $TS(A)$ and $abm(A)$

Consider a tool $A$ providing a $TS(A)$ of a drawing window with an associated $abm(A)$ consisting of window events such as MouseDown, MouseUp, MouseMoved, WindowExposed, etc.. Its abstract base machine $abm(A)$ is well-defined since it spans the usage of $A$ (when coupled with application state). The user is only aware of the drawing tool service $TS(A)$ and is unaware of the underlying $abm(A)$. Ideally, we desire to abstract $abm(A)$ so as to specify $TS(A_i) = abm(A_i)$. This allows us to specify semantical queries over the usage of an applet.

### Temporal-Awareness

Temporally-awareness relates time to the usage of tool services. A temporally-aware tool $A_i^+$ associates (during capture) and enforces (during playback) the notion of relative time to its abstract base machine, i.e., $TS^+(A_i, t) = f(abm(A_i), t)$.

### Replayable Applets

A replayable applet $A_i*$ is a temporally-aware tool, applet, or device driver with a well-defined tool service $TS(A_i)$.

### Replay-Awareness

As temporal-awareness is defined *wrt* a tool, replay-awareness is defined *wrt* the computer-based workspace spanned by those tools. A workspace that is spanned by replayable applets that comply with our replay-awareness interfaces is said to be replay-aware. These interfaces (see Fig. 7.2) allow a third-party, our session manager, to orchestrate multiple tools in a workspace as a collective. Replay-awareness allows the capture, representation, playback, and manipulation of multiple tools in the workspace.

### Applet Streams

Over time, the usage of a tool service is modeled by a temporal media stream, (referred to as *an applet stream* $a$), that consists of a time-ordered sequence of abstract base machine operations, i.e., $a = [(a_1 = f_i(), t_1), a_2 = (f_k(), t_2), \cdots, a_n = (f_j(), t_n)]$.

### Temporal-Awareness Interfaces

Two primitives are used to extend temporal-awareness over an applet stream. The record() primitive time-stamps and stores an event. The replay() primitive loads and schedules an event. Two additional interfaces are used to fine-tune the performance of the previous primitives: store() and prefetch(). The store() primitive is associated to the record() primitive and provides efficient, media-dependent, storage of stream events. The prefetch() primitive is associated the replay() primitive and provides efficient, media-dependent, prefetching of stream events.

### Stream Events

To support interactive, delay-sharing of a session, the record() and replay() primitives operate over four different types of events: (1) applet events $e_i$, (2) state snapshots $S_i$ (3) synchronization events $s_i$, and (4) branch events $b_i$. The first two types of events represent intra-applet events, i.e., events that are internally consumed by the applet. The later two

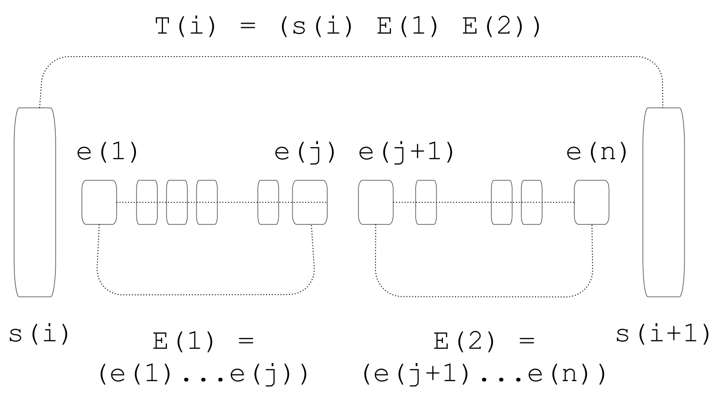


**Figure 7.3: Relationship between events $e_i$, event sequences $E_i$ synchronization events $s_i$, and scheduling intervals $T_i$.**

are inter-applet events, i.e., events that are consumed by multiple applets. The basic model of a temporal media stream is shown in Fig. 7.3. The designer of a replayable applet must specify the contents of the intra-applet events (i.e., applet events and state snapshots). Both applet events and state snapshots are defined by the selection of an abstract base machine *abm* for a given applet.

### Synchronization Events

The synchronization event $s_i(R(\cdots))$ enforces a time equals synchronization relationship across streams. Whenever an exact temporal ordering across streams in a relationship $R(\cdots)$ is needed, the synchronization event $s_i(R)$ is used. The frequency and placement of a synchronization request is defined by the application developer. A developer uses a synchronization event $s_i$ to enforce a synchronization relationship using relative point synchronization. The placement of synchronization checkpoints must occur at the end or beginning of a well-defined event sequence.

### State Snapshots

A state snapshot $S_i$ contains the full state of an applet and as such state snapshots are defined by the context of an applet. For example, the state of a drawing application is composed of the value for all variables of an applet (items such as pen color, canvas color, canvas size, pen size, etc.). Assume $S_0, ..., S_{i-1}, ...S_n$ is a sequence of state snapshots taken just after synchronization events $s_0, ..., s_{i-1}, ...s_n$.

### Branch Events

A branch event $b_j$ introduces a fork over the re-execution flow of an applet stream (see Fig. 7.4). Because applet streams are stateful, in order to support support editing and annotation of sessions, branch events need to be preceded by a state snapshot event $S_k$.

Fig. 7.4 further illustrates the supporting model for editing a previously recorded session. Editing of a re-executable segment relies on branching events $b_i$, state snapshots $S_i$. The original segment is shown as $A'B$. Interaction reusing the initial state to this segment is shown as $A''B$. Finally, annotation over this segment then coupled with a state jump is shown on by $A''C$. Equivalently,

$$T = \left\{ \begin{array}{l} A'B = (s_A, S_i, b_i(A'), E_1, E_2)s_B \\ A''B = (s_A, S_i, b_i(A''), E'_1, E'_2, b_i(B))s_B \\ A''C = (s_A, S_i, b_i(A''), E'_1, E'_2, b_i(C))s_C \end{array} \right\} \quad (7.1)$$

### Characterization of Heterogeneity

Applet streams have heterogeneous characteristics. Events from different applet streams (can) have different characteristics such as:

- event density:

  that is, are events coarse or very fine grained?

- event timing characterization:

  that is, are events asynchronous? In that case, could they be characterized by an inter-arrival distribution or does a periodical characterization describes them (as for continuous streams).

- event statefulness:

  that is, are events causally related or are events temporally related?

Such heterogeneity affects the suitability of known policies for synchronization and scheduling of streams.

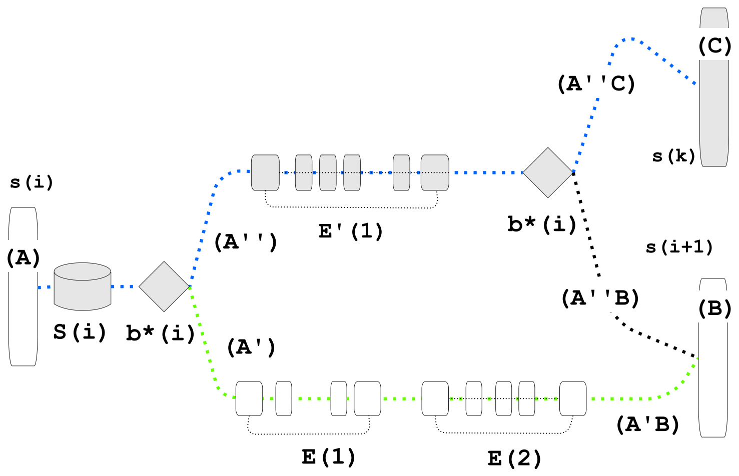


**Figure 7.4: Editing of a re-executable segment relies on branching events $b_i$, state snapshots $S_i$. The original segment is shown as A'B. Interaction reusing the initial state to this segment is shown as A"B. Finally, annotation over this segment then coupled with a state jump is shown on by A"C.**

### Event Sequences

An event sequence is a series of consecutive stream events. Event sequences must be well-defined. A well-defined event sequence represents an atomic, logical data unit, to the processing (i.e., storage, prefetching, scheduling, synchronization) of an applet stream.

### Well-Defined Event Sequences

In a well-defined sequence $E_i = (e_m, ..., e_n)$, intermediary events $(e_{m+1}, ..., e_n)$ lack stateful dependencies to events outside the sequence $E_i$. Event $e_m$ may be statefully dependent on its immediately preceding state snapshot. Event sequences represent the grain of our synchronization and scheduling mechanisms. This ensures that operations such as prefetching and synchronization are performed at well-defined, stateless, boundaries.

### Scheduling Interval

The scheduling interval provides the basic operational unit for our scheduling and synchronization mechanisms. A scheduling interval $T_i$ is composed of one or more event sequences $E_i$. Scheduling intervals are delimited at each end by synchronization events. Optional state snapshots may precede an event sequence. During playback, a branching events may be inserted immediately after a state snapshot (to support interactivity and editing of a session). Equivalently, the scheduling interval is formally specified by the following regular expression:

$$T_{i+1} = (s_i, \left\{ \begin{array}{l} [S_i], E_i \\ S_i, b_i, \left\{ \begin{array}{l} E_i \cdots \\ E_i' \cdots \end{array} \right\} \end{array} \right\}), s_{i+1} \tag{7.2}$$

### Synchronization Relationships

A synchronization relationship $R(a_i \leftarrow a_j)$ : *policy* characterizes the nature of the synchronization relationship between $a_i$ and $a_j$ in terms of (1) a master-slave dependency constraint (i.e., the playback of $a_j$ synchronizes to $a_i$); and (2) a treatment policy for the handling of the dependency constraint (i.e., the schedule of $a_j$ adapts to $a_i$). For example, the synchronization relationship $R(audio \leftarrow gestures)$ : *policy*, where *policy* prescribes "*gestures are to adapt to the unconstrained playback of audio*" represents an end-user characterization of the playback of audio-annotated gesturing.

### Types of Synchronization Relationships

There are two types of synchronization relationships between heterogeneous streams: **causal** and (purely) **temporal** (or non-causal). In a causal relationship, our goal is to preserve causality inherent across events of different streams. Causal relationships typically arise between discrete streams (e.g., cut and paste between two applet streams) although it can be found too, between a discrete and a continuous stream (e.g., user interactions over a video stream). In a **temporal relationship**, our goal is to preserve relative timing during the streaming across events from different streams. Purely temporal relationships typically arise between continuous streams (e.g., lip-synchronous video) as well as between a continuous and discrete streams (e.g., audio-annotated gesturing).

### Dependency Constraints

A synchronization event $s_i(R(a_1 \cdots a_m))$ is used to request point synchronization between scheduling intervals $T_i(a_1) \cdots T_i(a_m)$ of different applet streams involved in a synchronization relationship $R(a_1 \cdots a_m)$. A synchronization event $s_i(R(lhs \leftarrow rhs))$ is implemented as one or more pair-wise (binary) dependency constraints between each slave stream in the $rhs$ of $R(\cdots)$ (those on the right hand side of the arrow) and their master stream $lhs$ (the left hand side). For example, the synchronization relationship $R(a_j \leftarrow a_k a_m)$ (read as "*streams ($a_k$ & $a_m$) <u>synchronize to</u> stream $a_j$*") leads to the following two pair-wise dependency constraints:

$$\left\{ \begin{array}{l} d_{jk}(a_j \rightarrow a_k) \\ d_{jm}(a_j \rightarrow a_m) \end{array} \right\} \tag{7.3}$$

A dependency constraint $d_{jm}(a_j \rightarrow a_m)$ is read as "*stream $a_j$ <u>releases</u> stream $a_j$*" and specifies that when $a_m$ is ahead of stream $a_j$, stream $a_m$ is to "*meet*" with stream $a_j$ for each corresponding synchronization event across the two streams. As a result, whenever stream $a_m$ fails to "*meet*" $a_j$ at a synchronization event, then a synchronization treatment $treat$ is applied over $a_m$ ($a_m$ waits, when ahead of $a_j$).

Dependency constraints are asymmetrical (e.g., $a_j$ does not wait for $a_m$ when $a_j$ is ahead). The asymmetry of a dependency constraint $d_{jm}$ can be defined in terms of the playout times of synchronization events $s_i R(a_j \rightarrow a_m)$ as follows:

$$d_{jm}(a_j \leftarrow a_m) = \left\{ \begin{array}{l} t_{s_i(a_j)} > t_{s_i(a_m)} : treat(a_j); \\ t_{s_i(a_j)} \approx t_{s_i(a_m)} : nil; \\ t_{s_i(a_j)} < t_{s_i(a_m)} : nil; \end{array} \right\} \tag{7.4}$$

where $treat$ specifies the synchronization treatment to be applied over $a_m$ for correcting the playback asynchrony between $a_j$ and $a_m$ (that is, corrected only when $a_m$ is ahead of $a_j$). We refer to this scenario as $a_m$ failing to "*meet*" its dependency constraint with $a_j$. The notion of a pair-wise "*meet*" of streams is central to our approach and it is discussed below.

### Synchronization Policies and Treatments

Different synchronization relationships specify different ways of satisfying the "*meet*" constraint through the *policy* specification. Let $i$ be the $i^{th}$ synchronization event of an applet stream $a_j$ and $i^*$ be its matching (i.e., $i^{th}$) synchronization event on another applet stream $a_m$. There are three basic meeting policies that specify how $i$ and $i^*$ *ought to meet* during the playback of these two applet streams. These are:

- absolute: $i = i^*$

  that is, the synchronization points $i$ and $i^*$ must be replayed at the same time.

- lazy: $i - \epsilon < i^* < i + \epsilon$

  that is, the synchronization points $i$ and $i^*$ can be replayed relatively close to each other.

- laissez-faire: $k = i^*$

  that is, the synchronization points $i$ and $i^*$ can be replayed in any sequence.

A *policy* specification for a relationship specifies one of the aboves options. A treatment *treat* represents the run-time enforcement of a pre-selected policy.

### Strongest Synchronization Policy

Not all these "*meet*" policies are the same. Meet policies can be ordered in terms of the strength of their tolerance of asynchrony departure between synchronization points $i$ and $i^*$. Equivalently,

$$\text{absolute} > \text{lazy} > \text{laissez-faire} \tag{7.5}$$

### Types of Dependency Constraints

Between any two applet streams $a_j$ and $a_m$, there are three different types of dependency constraints: 2-way, 1-way, and 0-way. In a 2-way dependency constraint, stream $a_j$ synchronizes to stream $a_m$ *viceversa.* Whenever either stream is ahead of the other, it waits and "*meets*" the other. The notation $(j : m)$ is a short-hand (referred to as a 2-way set or informally as a *tight synchronization pair*) for the representation of the 2-way constraint $[a_j \rightarrow a_m, a_j \rightarrow a_m]$. In a 1-way dependency constraint $d_{jm}(a_j \rightarrow a_m)$, stream $a_m$ synchronizes to stream $a_j$. Whenever $a_m$ fails to "*meet*" (is ahead of) $a_j$, $a_m$ waits for $a_j$. Finally, in a 0-way dependency constraint, neither stream synchronizes to the other. That is, streams $a_j$ and $a_m$ are unrelated; neither one waits.

### Relationships Between Multiple Streams

The specification of a workspace $W$ may result in possibly, $n$ synchronization relationships $R_{k=1 \cdots n}$. In turn, each synchronization relationship can be specified by as many as $n$ pair-wise dependency constraints. An $n$-ary synchronization handshake (see Section

8.5.1) is used by the session manager to enforce the up-to $n$ dependency constraints of an arbitrary relationship $R(a_1 \cdots a_n)$. A constraint/update propagation matrix (see Section 8.5.2) is used to enforce the $n$ synchronization relationships between streams.

### Dependency Preservation Mechanism

Not all synchronization relationships have similar tolerances — for example, consider the temporal specification for lip-synchronous video vs. that required for audio-annotated slide-shows. A mechanism is needed to preserve the heterogeneity of co-existing synchronization relationships.

Inside an applet, synchronization relationships are preserved as follows. During design of a workspace, its designer needs to specify the various synchronization relationships that relate each applet *wrt* other applets in the workspace. To enforce a given synchronization relationship (say $R(lhs \leftarrow rhs)$), the developer of an applet uses synchronization events as follows. During the record of a session, the $lhs$ applet publishes a synchronization event $s_i(R(lhs \leftarrow rhs))$ to the session manager. Then, the session manager introduces a dependency constraint into each stream on the $rhs$ of $R(\cdots)$ by inserting a synchronization event $s_i(R(\cdots))$ into each of them. A synchronization event is logged as being published by $lhs$ and subscribed by $rhs$ streams. During playback, on processing the synchronization event $s_i(R(\cdots))$ both $lhs$ as well as $rhs$ streams report to the session manager which then decides the proper (pair-wise) run-time treatment for each of the $rhs$ streams (if any) given the *policy* associated with this $R(\cdots)$.

### Handling of Conflicting Relationships

Since multiple $n$-ary relationships can co-exist, it is possible for a given stream to hold multiple relationships under possibly, different policies. When a conflict between different policies arises, the *strongest policy* in the conflict is selected. The handling of a conflict between policies of equal strength is discussed in Section 7.5.

### Specification of Replayable Workspaces

The specification of a workspace $W$ produces three tuples: (1) $W_A$: applets in the workspace, (2) $W_R$: relationships between applet streams, and (3) $W_T$: playback tolerances of each relationship.

## 7.4 A Specification Example

A replayable workspace is designed at two levels: at the applet level and at the workspace level. The developer of a replayable applet is a programmer who retrofits replay-awareness to an applet by complying with our replay-awareness interfaces. On the other hand, the workspace designer is a system modeler who deals with the specification of a workspace. For example, the workspace $W = \{W_A, W_R, W_T\}$ on Fig. 7.5 could be specified as:

$$W = \left\{ \begin{array}{l} W_A = (A, B, C, D); \\ W_R = (R_1(A : B), R_2(C \leftarrow A), R_3(C : D)); \\ W_T = (causal, temporal, temporal); \end{array} \right\} \tag{7.6}$$

The playback of a workspace requires the preservation of heterogeneous fine-grained relationships between applet streams. For example, the playback of $W$ requires the preservation of three fine-grained synchronization relationships between applet streams (see Fig. 7.5).

**R1:** a 2-way causal relationship;

- A and B are discrete streams with strong temporal correspondence.
- goal: achieve smooth playback of both streams.
- constraint: preserve causality between events across streams. Causality here is bi-directional across streams (A to B) and (B to A).

**R2:** a 1-way temporal relationship;

- A is discrete and C is continuous with relaxed temporal correspondence.
- goal: achieve smooth playback of A and playback continuity on C.
- constraint: preserve relative timing during the streaming between inputs from A and C. This relationship is not causal.

**R3:** a 2-way temporal relationship;

- C is continuous and D is discrete with relaxed temporal correspondence.

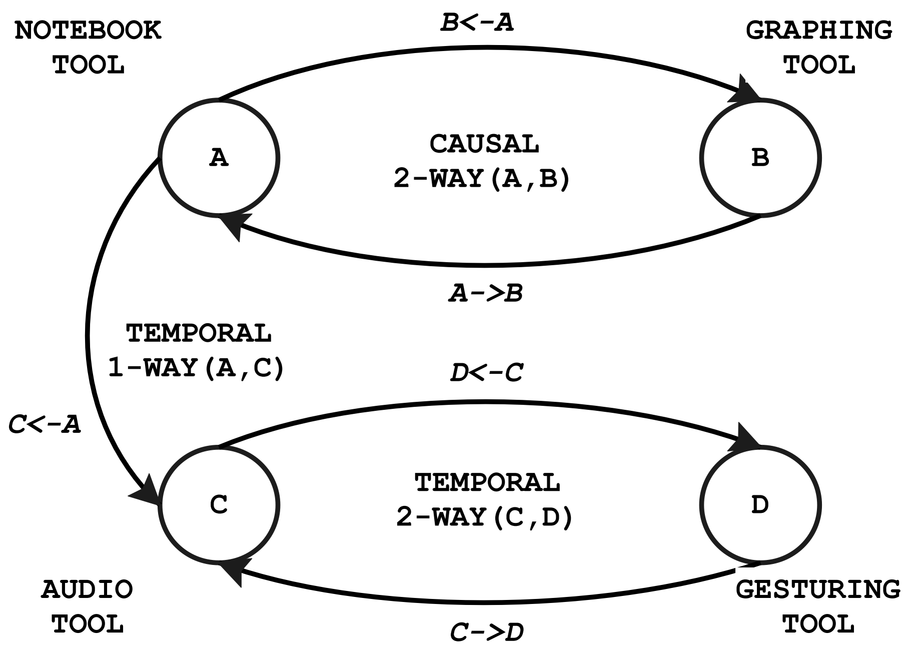


Figure 7.5: Relationships between events of different applets may be different. Playback of the workspace requires preserving these relationships. For example, the playback of the notebook and graphing tools requires enforcing a causal relationship (such as cut and paste) between them. On the other hand, the playback of audio-annotated pointer gesturing requires only a temporal (non-causal) relationship. Faithfully playback of the above workspace requires preserving the following synchronization relationships: (R1) a causal 2-way relationship in the streaming of inputs to A and B; (R2) a temporal 1-way relationship to adapt the streaming of inputs of A to the progress of C; and (R3) a temporal 2-way relationship in the streaming of inputs to C and D.

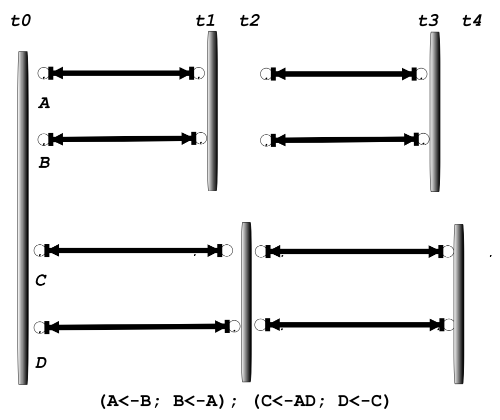


**Figure 7.6: Here, the A:B pair re-executes faster than the $C : D$ pair. Each pair is kept synchronized by independent 2-way relationships $R_1(A : B)$ and $R_3(C : D)$. However, the 1-way relationship $R_2(C \leftarrow A)$ forces the $A : B$ pair to synchronize to the $C : D$ pair. Playback continuity on the $C : D$ pair is unaffected while the $A : B$ pair observes a discontinuity of duration $t_2 - t_1$.**

- goal: achieve playback continuity on C and smooth playback of D.
- constraint: preserve relative timing during the streaming between inputs from C and D. This relationship is not causal.

Relationship R1 arises from the need to support causal dependencies such as "cut and paste" between notebook and graphing tools. R2 arises from the need to preserve temporal correspondence of audio-annotations over the playback of the notebook. Finally, R3 arises from the need to preserve temporal correspondence of audio-annotated pointer-gesturing.

### 7.4.1 Continuity/Synchronization Tradeoffs Between Relationships

In the previous specification example, a cautious reader should have note that, under apparently identical circumstances, R2 is 1-way relationship while R3 is a 2-way relationship. This is a design decision; the sort of decision typically made by the designer of a workspace. Let's analyze such decision. Throughout this discussion, recall that a a tight synchronization pair is a 2-way set between two streams.

During playback, whenever the 2-way set $(A : B)$ is ahead of the 2-way set $(C : D)$, the 2-way set $(A : B)$ attempts to "*meet*" with $C$ and in this case, with the 2-way set $(C : D)$ — a result of its 1-way relationship $R(C \leftarrow A)$. In such cases, the 2-way set $(A : B)$ is forced

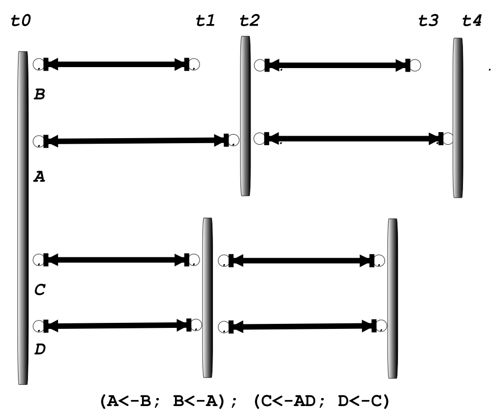


**Figure 7.7: Here, notebook (A) and graphing (B) tools re-execute slower than audio (C) and gesturing (D) tools. While 2-way relationships preserve synchronization on $A : B$ and $C : D$, the 1-way relationship $R_2(C \leftarrow A)$ allows a relaxation of the temporal correspondence between $A : B$ and $C : D$ by $(t_2 - t_1)$. Synchronization between (*relationship*) pairs is traded-off for continuity on the $C : D$ pair.**

to wait for the $(C : D)$ set. Here, the 1-way relationship favors continuity on the playback of the 2-way set $(C : D)$ to continuity on the 2-way set $(A : B)$ while it preserves tight synchronization between the two 2-way sets under this condition. Fig. 7.6 illustrates this condition; playback continuity on the 2-way set $(C : D)$ is unaffected during time $[t_1 \cdots t_2]$ by what happens with the 2-way set $(A : B)$. However, the 2-way set $(A : B)$ is forced to wait for the 2-way set $(C : D)$ during time $(t_2 - t_1)$. Synchronization between the two 2-way sets is achieved at time $t_2$ at the expense of a discontinuity $(t_2 - t_1)$ on the 2-way set $(A : B)$. If an underlying asynchrony exists between the two 2-way sets, a long-term scheduling compensation over one of the 2-way sets might be desirable.

On the other hand, whenever the 2-way set $(A : B)$ is behind the 2-way set $(C : D)$, the 2-way set $(A : B)$ continues — a result of the asymmetry of the 1-way relationship $R(C \leftarrow A)$. Again, this decision favors continuity on the playback of the 2-way set $(C : D)$ by relaxing the synchronization between these two 2-way sets. Fig. 7.7 illustrates this condition; playback of the 2-way set $(A : B)$ is not disrupted by $C$ at time $t_1$. Again, playback continuity of the 2-way set $(C : D)$ remains unaffected. Playback continuity of the 2-way set $(A : B)$ is also unaffected. Synchronization between the 2-way sets $(A : B)$ and $(C : D)$ is relaxed by time $(t_2 - t_1)$. Finally, the temporal correspondence inside each

2-way set is kept tight. That is, synchronization between the two 2-way sets is traded-off for continuity on each 2-way set. If an underlying asynchrony exists between the two 2-way sets, a long-term scheduling compensation over one of the 2-way sets may be necessary.

This example provided insight into the sort of tradeoffs that can be *specified* between continuity and synchronization across multiple heterogeneous $n$-ary *relationships* (as opposed to *streams*). In the next section, we analyze the specification of tradeoffs within a relationship (i.e., $n$-ary dependency constraints).

## 7.5 Generalized Specification of Heterogeneous Relationships

A workspace $W$ of $n$ heterogeneous streams is specified by a tolerance matrix $W_T$, which provides a compact characterization of all the relationships between streams in the workspace as follows:

$$W_T = \left\{ \begin{array}{cccccc} -1 & d_{12} & \cdots & d_{1j} & \cdots & d_{1n} \\ d_{21} & -1 & \cdots & d_{2j} & \cdots & d_{2n} \\ \cdots & \cdots & \ddots & \cdots & \cdots & \cdots \\ d_{i1} & d_{i2} & \cdots & d_{ij} & \cdots & d_{in} \\ \cdots & \cdots & \cdots & \cdots & \ddots & \cdots \\ d_{n1} & d_{n2} & \cdots & d_{nj} & \cdots & -1 \end{array} \right\} \tag{7.7}$$

where $d_{ij}$ is a short-hand for $d(i \rightarrow j)$ — that is, $a_i$ releases $a_j$ when $a_j$ is ahead. This corresponds to the synchronization relationship $R(a_i \leftarrow a_j)$. Each entry $d_{ij}$ of $W_T$ characterizes the asymmetrical relationship between $i$ and $j$ in terms of a "*meet*" constraint as follows:

$$d_{ij} = \left\{ \begin{array}{lcl} \sigma_{ij} & : & for\ i \neq j; \\ -1 & : & for\ i = j; \end{array} \right\} \tag{7.8}$$

A "*meet*" constraint $d_{ij} = \sigma_{ij}$ constrains the playback of two streams $i$ and $j$ in terms of:

- a dependency constraint $d_{ij}$.
- the tolerance $\sigma_{ij}$ of the dependency constraint $d_{ij}$ to asynchrony $\epsilon^*_{ij}$ — the estimation of the asynchrony is further discussed in **Appendix A**.

Note that the diagonal elements $d_{ii}$ are set to $-1$. This justs flags the fact that no "*meet*" constraint exists between a stream and itself. In all other cases, the "*meet*" constraint is specified through a single parameter $\sigma_{ij}$. This parameter is referred to as the "*tolerance of j to asynchrony from i*".

This scheme allows flexible enforcement of "*meet*" constraints that allows us to address heterogeneity on synchronization relationships. Our three treatment policies (absolute, lazy, and laissez-faire) can be expressed by this framework by characterizing the corresponding $\sigma_{ij}$ values as follows:

$$\sigma_{ij} = \left\{ \begin{array}{lcl} 0 & : & absolute \\ 0 < c < \infty & : & lazy \\ \infty & : & laissezfaire \end{array} \right\} \tag{7.9}$$

The tolerance matrix $W_T$ captures the asymmetry of 1-way relationships as the coupling of an unconstrained *laissez-faire* and a dependency constraint. This asymmetry is observed in transposable entries as:

$$1 - way : \left\{ \begin{array}{c} d_{ij} = \infty \\ 0 \leq d_{ji} < \infty \end{array} \right\} \tag{7.10}$$

The tolerance matrix $W_T$ also captures the symmetry of 2-way relationships as the coupling of two identical dependency constraints. This symmetry is observed in transposable entries as follows.

$$2 - way : \left\{ \begin{array}{c} 0 \leq d_{ij} < \infty \\ 0 \leq d_{ji} < \infty \end{array} \right\} \ \& \ d_{ij} = d_{ji} \tag{7.11}$$

Fig. 7.8 illustrates the modeling difference between 1-way and 2-way relationships *wrt* two streams $a$ and $b$. Again, consider the workspace in Fig. 7.5. Such workspace can be specified by the following tolerance matrix:

$$W_T(A,B,C,D) = \left\{ \begin{array}{cccc} -1 & \sigma_{ab} = 0 & \infty & \infty \\ \sigma_{ba} = 0 & -1 & \infty & \infty \\ \sigma_{ca} = k_1 & \infty & -1 & \sigma_{cd} = k_2 \\ \infty & \infty & \sigma_{dc} = k_2 & -1 \end{array} \right\} \tag{7.12}$$

where $k_1$ is a tolerance specification. The tolerance matrix $W_T$ provides a precise and concise representation of the workspace specification $W = \{applets, dependencies, tolerances\}$ as introduced in the previous section. It also provides a sound foundation for the analysis of the properties of these replayable workspaces.

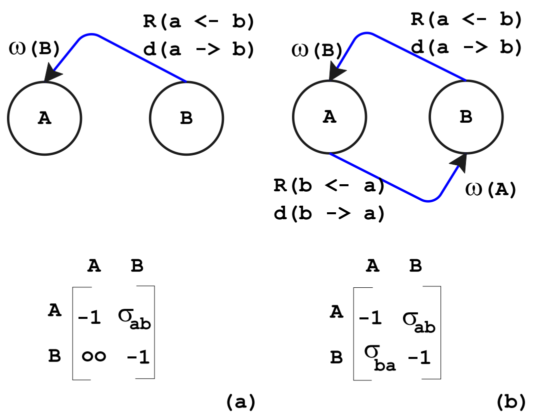


**Figure 7.8:** 1-way (left) and 2-way **relationships between two streams and associated $W_T$ matrices.**

Finally, the transpose of the tolerance matrix is the **constraint matrix** ($W_C$). Equivalently,

$$W_C = W_T^t = \left\{ \begin{array}{cccccc} -1 & c_{12} & \cdots & c_{1j} & \cdots & c_{1n} \\ c_{21} & -1 & \cdots & c_{2j} & \cdots & c_{2n} \\ \cdots & \cdots & \ddots & \cdots & \cdots & \cdots \\ c_{i1} & c_{i2} & \cdots & c_{ij} & \cdots & c_{in} \\ \cdots & \cdots & \cdots & \cdots & \ddots & \cdots \\ c_{n1} & c_{n2} & \cdots & c_{nj} & \cdots & -1 \end{array} \right\} \tag{7.13}$$

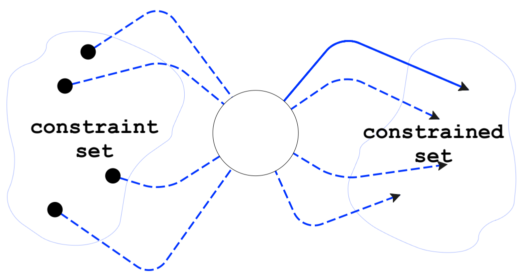
where $c_{ij} = d_{ji}$; a short-hand for $d(j \rightarrow i)$. The constraint matrix $W_C$ is useful to facilitate the visualization of the behavior of the constraint propagation algorithms.

### 7.5.1 Correctness of Heterogeneous Media Integration

The handling of heterogeneous relationships requires predictable integration of $n$-ary dependency constraints.

Fig. 7.9 shows how an arbitrary stream $j$ perceives its relationships to other streams on the workspace. Stream $j$ perceives the workspace to be divided in two sets: a constraint-set and a constrained-set. The constraint-set represents as the set of all streams that impose a "*meet*" constraint over the release of $j$. The constrained-set represents the set of all streams over which $j$ imposes a "*meet*" constraint. To applets, the workspace appears constraint-based and atemporal.

The session manager handles integration of its $n$ heterogeneous media streams through



.

**Figure 7.9: The view of the constraint handling problem as seen by an arbitrary stream $j$. The stream $j$ sees the workspace as divided into two sets: (1) constraint-set — the set of streams that impose a "*meet*" constraint over the release of $j$, that is, $i$ *such that* $(0 \leq d_{ij} < \infty)$. (2) constrained-set — the set of streams over which $j$ imposes a "*meet*" constraint, that is, $k$ *such that* $(0 \leq d_{jk} < \infty)$. These sets are not mutually exclusive.**

periodical point-based constraint resolution. The workspace appears to the session manager as being temporally-based but in a rather chaotic state in terms of constraint management. Fig. 7.10 illustrates the sort of temporal chaos perceived by the session manager during an arbitrary scheduling interval of $n$ streams.

Next, we argue the correctness of our heterogeneous media integration for an arbitrary stream $j$ with respect to both its constraint and its constrained sets. First, let's sort the temporal view of an arbitrary scheduling interval by increasing "finish" times for each of the $n$ individual scheduling intervals. The result should be similar to that shown in Fig. 7.11.

Media integration relies on "*meet*" constraints that are evaluated for every synchronization dependency in the constraint set of $j$ (see Fig. 7.12). The release of $j$ is constrained by each "*meet*" constraint in this set. However, the parameter $\sigma_{ij}$ relaxes the "*meet*" constraint of $c_{ji}$ by tolerating some asynchrony between $j$ and $i$. During run-time, a smoothed indicator of the asynchrony between these two streams $\epsilon^*_{ji}$ is computed. Note that $\epsilon^*_{ij}$ is negative when $j$ ahead of $i$. A "*meet*" constraint is satisfied when $-\sigma_{ij} < \epsilon^*_{ji}$.

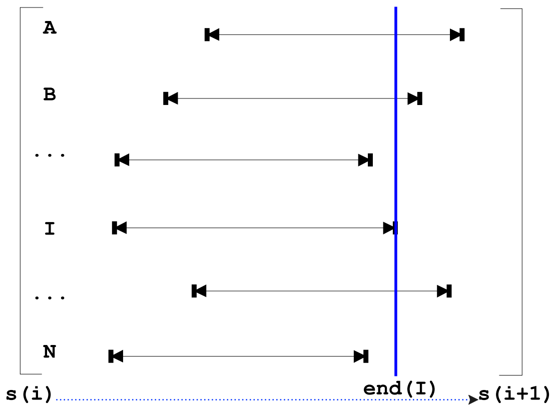


Figure 7.10: Temporal view to constraint handling on an arbitrary scheduling interval across $n$ streams $(A \cdots I \cdots N)$. The lenght of the $i^{th}$ horizontal line segment represents the relative playout duration of the scheduling interval on the $i^{th}$ stream. Playout duration is unpredictable for re-executable content.

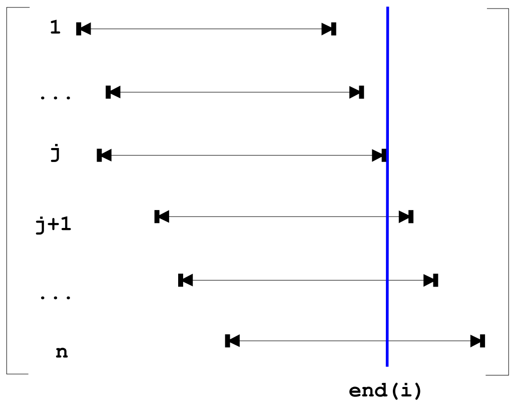


Figure 7.11: Temporal sort of the scheduling interval by increasing "finish" times.

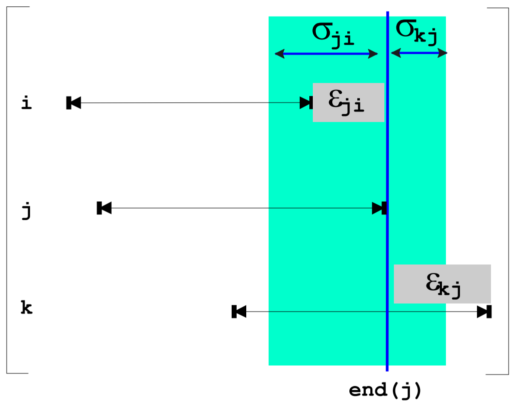


**Figure 7.12: The release of $j$ is constrained by all the "*meet*" constraints on its constraint set. However, the parameter $\sigma_{ij}$ relaxes a "*meet*" constraint $c_{ji}$ by tolerating some asynchrony between streams $j$ to $i$. During run-time, such asynchrony $\epsilon^*_{ji}$ is estimated. A "*meet*" constraint is satisfied when $-\sigma_{ij} < \epsilon^*_{ji}$.**

### The "*meet*" Constraint

A "*meet*" constraint $c_{ji}(a_j \rightarrow a_i) = \sigma(a_i, a_j)$ between streams $a_j$ and $a_i$ is satisfied when the asynchrony indicator $\epsilon^*_{ij}$ is estimated to be within statistically tolerable limits (say $\sigma_{ij}$). Equivalently,

$$meet(a_i, a_j) = \left\{ \begin{array}{ll} YES & -\sigma_{ij} < \epsilon^*_{ji} \\ NO & otherwise \end{array} \right\} \tag{7.14}$$

### The Constrained Set of a Stream

The constrained (OUT) set of a stream $a_i$ is composed of nodes $a_k$ connected by a direct outgoing edge from $a_i$ in $W_C$ — note that it says $W_C$ not $W_T$. Equivalently,

$$OUT(a_i) = a_k \text{ such that } (0 \leq d_{ik} < \infty) \tag{7.15}$$

Synchronization-wise, a stream $a_k \in OUT(a_i)$ has a dependency constraint $d_{ik}$ with $a_i$. At every scheduling interval, when $a_k$ is ahead of $a_i$, $a_k$ "*meets*" $a_i$ before continuing. Scheduling-wise, $a_k$ adapts its playback to $a_i$. For example, consider the workspace specification on Equation 7.12 where $OUT(A) = \{B\}$; $OUT(B) = \{A\}$; $OUT(C) = \{A, D\}$; and $OUT(D) = \{C\}$.

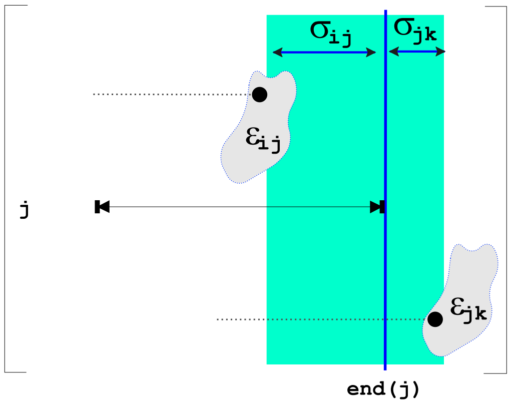


**Figure 7.13: The coupling of the temporal ordering and the "*meet*" constraint during the handling of multiple 1-way temporal relationships. Two disjoint sets are created during run-time over the temporal view of a scheduling interval: the set of streams that completed significantly before $j$ (positive outliers for an $\epsilon_{ij}$); and the set of streams that complete significantly after $j$ (negative outliers for an $\epsilon_{kj}$).**

### The Constraint Set of a Stream

The constraint set (IN) of a stream $a_i$ is composed of all incoming edges from some $a_j$ to $a_i$ in $W_C$. Equivalently,

$$IN(a_i) = a_j \text{ such that } (0 \leq d_{ji} < \infty) \tag{7.16}$$

Synchronization-wise, a stream $a_j \in IN(a_i)$ imposes a dependency constraint $d_{ji}$ over $a_i$ — respectively, $R(a_j \leftarrow a_i)$ holds. At every scheduling interval, when $a_i$ is ahead of $a_j$, $a_i$ must "*meet*" $a_j$ before continuing. Scheduling-wise, $a_i$ adapts its playback to the $IN(a_i)$. For example, consider the workspace specification on Equation 7.12 where $IN(A) = \{BC\}$; $IN(B) = \{A\}$; $IN(C) = \{D\}$; and $IN(D) = \{C\}$.

### The Run-Time Outlier Subset of $IN$

The outlier set $IN^*(a_i)$ is the *run-time subset* of streams in $IN(a_i)$ for which the asynchrony indicator $\epsilon^*(a_j, a_i)$ exceeds its tolerance $\sigma(a_j, a_i)$. Equivalently,

$$IN^*(a_i) = a_j \in \{IN(a_i) \text{ such that } (|\epsilon^*(a_j, a_i)| > \sigma(a_j, a_i))\} \tag{7.17}$$

**The Run-Time Outlier Subset of $OUT$**

The outlier set of $OUT^*(a_i)$ is the *run-time* subset of streams in $OUT(a_i)$ for which the asynchrony indicator $\epsilon^*(a_i, a_k)$ exceeds its target tolerance $\sigma(a_i, a_k)$). Equivalently,

$$OUT^*(a_i) = a_k \in \{OUT(a_i) \text{ such that } (|\epsilon^*(a_i, a_k)| > \sigma(a_i, a_k))\} \tag{7.18}$$

**Discussion**

A run-time outlier subset of a stream $a_i$ represents any stream $a_j$ having either a dependency constraint $d_{ij}$ (for $OUT(a_i)$) or $d_{ji}$ (for $IN(a_i)$) with stream $a_i$ for which (during run-time) a statistically significant asynchrony departure is observed (see Fig. 7.13).

**Negative Outliers**

A negative outlier set $S^-(a_i)$ is the subset of streams in an outlier set $S^*(a_i)$ for which the asynchrony indicator is negative. Equivalently,

$$S^-(a_i) = a_j \in \{S^*(a_i) \text{ such that } (\epsilon^*(a_i, a_j) \leq -\sigma(a_i, a_j)\} \tag{7.19}$$

**Positive Outliers**

A positive outlier set $S^-(a_i)$ is the subset of streams in an outlier set $S^*(a_i)$ for which the asynchrony indicator is positive. Equivalently,

$$S^+(a_i) = a_j \in \{S^*(a_i) \text{ such that } (\epsilon^*(a_i, a_j) > \sigma(a_i, a_j))\} \tag{7.20}$$

**Run-Time Scheduling Conflicts**

We define a run-time scheduling conflict over a stream $j$ as a temporal view applied over a tolerance matrix for which distinct statistically significant trends exist. There are two cases to consider.

- magnitude conflict:

  A magnitude conflict occurs when an outlier set of $OUT^*(a_j)$ is not empty and contain two or more distinct (statistical significant) trends.

- sign conflict:

  A sign conflict occurs when both positive and negative outliers of $OUT^*(a_j)$ are not

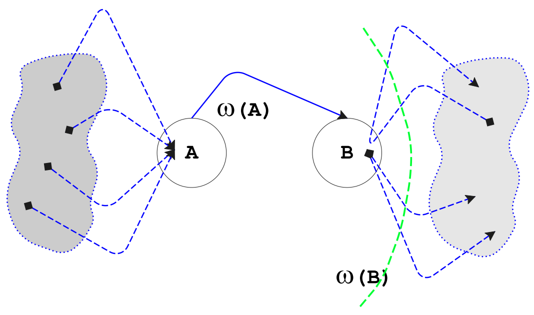


**Figure 7.14: A synchronization conflict is possible when $\|IN()\| > 1$ (see A). A scheduling conflict is possible when $\|OUT()\| > 1$ (see B).**

empty and each contains one or more (statistical significant) trends.

**Example 1:** Let $a, b, c$ be streams for which the synchronization relationships $R(b \leftarrow a)$ & $R(c \leftarrow a)$ hold. A run-time (sign-based) scheduling conflict for $a$ arises in any scheduling interval whenever the following condition is met.

$$|\epsilon^*_{ba}| > \sigma_{ba} \ \& \ |\epsilon^*_{ca}| > \sigma_{ca} \ \& \ \frac{\epsilon^*_{ba}}{\epsilon^*_{ca}} < 0 \qquad (7.21)$$

The conflict arises because $a$ requires a positive scheduling compensation over $R(b \leftarrow a)$ while a negative scheduling compensation is required over $R(c \leftarrow a)$.

Scheduling conflicts produce at worst $r = \|IN^*(i)\|$ distinct pair-wise schedule compensations $\delta_{ji} = f(\epsilon_{ji}, \sigma_{ji})$ during run-time. In general, there are three possible ways to combine such pair-wise schedule compensations into one global schedule compensation $\hat{\delta}_j$.

- $\hat{\delta}_i = \delta_{ji}$; — for *a given* j.
- $\hat{\delta}_i = f(\delta_{ji})$; — for *some* smoothing function $f()$.
- $\hat{\delta}_i = 0$; — i.e., select none.

It is unclear whether choosing the largest, smallest, $k^{th}$ largest, mean, median, etc. produces a globally optimal compensation — or even whether one such exist. Across scheduling intervals, the first two policies could easily over- as well as under-compensate streams in $IN^*(i)$. This limits our ability to handle, with predictable behavior, arbitrary $n$-ary dependencies to only those cases where $r \leq 1$; that is, when the $IN(i)$ has only one temporal relationship. However, through the coupling of scheduling with synchronization measures associated with temporal 2-way relationships it is possible to extend predictable behavior to the handling of a special case of $n$-ary dependencies.

**Example 2:** Let $a, b, c, d$ be streams for which the synchronization relationships $R(a : b)$ & $R(c : d)$ holds. Any tight synchronization pair $(i : j)$ (any 2-way relationship) is by definition a **strongly connected component** ($SCC$) — as dependencies edges leave and come from either of its two nodes. For example, streams $a$ & $b$ form $SCC(a, b)$. Similarly, $c$ & $d$ form $SCC(c, d)$. $SCC$s guarantee tight synchronization *wrt* all its members. However, these $SCC$s are unrelated. Consider now, adding a temporal 1-way relationship $R(c \leftarrow a)$; that is, $a$ adapts to $c$, implemented through a dependency constraint $d_{ac}(a \rightarrow c)$. This new edge has the effect of slaving the $SCC(a, b)$ to the $SCC(c, d)$ (as shown in Figs. 7.6 and 7.7).

**Example 3:** Let $a, b, c, d$ be streams for which the synchronization relationships $R(a : b)$ & $R(c : d)$ holds. Consider adding a 2-way synchronization relationship $R(a : c)$. This has the effect of merging the two original $SCC$s into a single one: $SCC(a, b, c, d) = \{SCC(a, b) + SCC(c, d)\}$. Such $SCC$ guarantees tight synchronization *wrt* all the streams.

The two previous examples illustrate two fundamental ideas. First, $SCC$s provide the basis for tight synchronization (a predictable behavior by definition) and second, a temporal 1-way synchronization relationships can be used to couple two $SCC$s in such a way so as to achieve predictable behavior. For any stream on a given $SCC$, a scheduling measure might be desirable but it is not required (as synchronization measures guarantee tight synchronization). This observation allows us to achieve predictable behavior during the handling of the following $n$-ary dependency constraints. Predictable behavior is possible for an $IN(j)$ set for which exactly $\|IN(j)\| - 1$ streams have a 2-way relationship with $j$. Such streams form a $SCC$ with $j$. Scheduling conflicts can never arise (since the schedule compensation of $j$ would be determined by only one stream). For example, consider node $A$ on Fig. 7.5 of Chapter 7 — the $IN(A) = \{B, C\}$, where $B$ has a 2-way relationship with $A$ (that is $(A : B)$). Under this setup, only $A$ adapts to $C$.

An $SCC$ can adapt its playback, too. The key problem for adaptable $SCC$s is a problem referred to as the convergence of its members. A scheduling conflict is different from the convergence problem discussed next in Chapter 8. Scheduling conflicts arise when a stream attempts to enforce more than two or more temporal 1-way relationships. On the other hand, the convergence problem arises when two streams form an adaptable tight synchronization pair (an $SCC$) of exactly two temporal 1-way synchronization relationships causing each stream to adapt to the other. Both problems, however, share similar solutions. The approach is to preempt the existence of the problem. For example, we can preempt the convergence problem by allowing only one member of an adaptable tight synchronization pair to adapt. Equivalently, for any stream $j$ in a $SCC$, we allow only one edge in $IN(j)$ to be related to a temporal 1-way relationship. As a result, for any stream in a $SCC$, no stream adapts to more than one neighbor so the convergence problem is avoided.

### Run-Time Synchronization Conflicts

We define a run-time synchronization conflict over a stream $j$ as a temporal view over an associated tolerance matrix that exhibits statistically significant trends for some streams to fall behind $j$ and, for others, to fall ahead of $j$. It follows that a run-time synchronization conflict on $a_i$ occurs only when both positive and negative outliers of $IN^*(a_j)$ are not empty.

**Example 4:** Let $a, b, c$ be streams for which the synchronization relationship $(a \leftarrow bc)$ holds. A run-time synchronization conflict for $a$ arises in any scheduling interval whenever the condition is met.

$$|\epsilon^*_{ba}| > \sigma_{ba} \ \& \ |\epsilon^*_{ca}| > \sigma_{ca} \ \& \ \frac{\epsilon^*_{ba}}{\epsilon^*_{ca}} < 0 \tag{7.22}$$

Such conflict arises because $a$ should wait for $b$ to catch-up while on the other hand, it should continue too, so as to keep up with $c$.

Synchronization conflicts do not represent a problem. Stream $a_j$ simply waits for streams in its positive outlier $IN^+(a_j)$ so as to satisfy its pair-wise "*meet*" constraints while on the other hand, because of the asymmetry of the "*meet*" constraint, no action is needed for streams in the negative outlier $IN^-(a_j)$.

```
$MediaLayer()\{$
    $OK = WorkspaceLayer();$

    $while(OK)\{$
        $j = GetNextUpdate();$
        $UpdateConstraintsWith(j);$
        $HandleUpdateFor(j);$
        $OK = WorkspaceLayer();$
    $\}$
$\}$
```

**Table 7.1: The** $MediaLayer()$ **Algorithm.**

```
$HandleUpdateFor(j)\{$
    $MeetWithSet(j, IN^*(j));$
    $AdaptToSet(j, IN^*(j));$
    $ReleaseStream(j);$
$\}$
```

**Table 7.2: The** $HandleUpdateFor()$ **Algorithm.**

```
$MeetWithSet(j, IN^*(j))\{$
    $\forall_i \in IN^-(j)$
        $meet(i, j);$
$\}$
```

**Table 7.3: The** $MeetWithSet()$ **Algorithm.**

```
$AdaptToSet(j, IN^*(j))\{$
    $if(\ \overline{\|IN^-(j)\| \& \|IN^+(j)\|}\ )$
        $adapt(max(|IN^*(j)|);$
$\}$
```

**Table 7.4: The** $AdaptToSet()$ **Algorithm.**

### 7.5.2 The *MediaLayer*() Algorithm

Tables 7.1 through 7.4 contain the specification of the media integration algorithm for $n$-ary dependency constraints — when restricted as discussed in the previous section. The *MediaLayer* algorithm specified in Tables 7.1 performs four simple tasks: (1) wait for the next Update() from a stream $j$, (2) propagate the update to the global constraint resolution, (3) update the $n$-ary dependency resolution for $j$, and (4) couple with processing at the workspace layer. The *MediaLayer*() algorithm takes $O(n)$ time per update — since step 1 takes $O(1)$ time, step 2 takes $O(n)$ time, step 3 takes $O(n)$ time, and step 4 takes $O(1)$ time — for each of its $n$ streams thus resulting in a total worst case time of $O(n^2)$.

The algorithm *HandleUpdateFor*() in Table 7.2 handles constraint resolution for the $n$-ary dependency constraint problem of stream $j$. After all the dependency constraints for stream $j$ are met, a check is made to determine whether the schedule of stream $j$ should be adapted or not. Then, stream $j$ is released. The specification of the *MeetWithSet*() algorithm in Table 7.3. A stream $j$ will not be released until all its dependency constraints are met since the *MeetWithSet*() algorithm returns only when all dependencies $d_{ij}(i \rightarrow j)$ are met. The proof follows from the definitions of the $meet(i, j)$ constraint and from the definition of the negative outlier of $IN^*(J)$. The implementation of these algorithms is specified in Appendix H.

## 7.6 Concluding Remarks

Although the characterization of workspace seemed similar to that of multimedia authoring environments, I showed the temporal relationships found in replayable workspaces to be different — I showed them to be of fine-granularity and strongly heterogeneous. In this chapter, I characterized the playback of tools on a workspace in terms of multiple, coexisting, heterogeneous temporal relationships. I presented a temporal model for the specification and modeling of replayable workspaces in terms of these temporal relationships and addressed their integration during playback. I showed these temporal relationships to be richly heterogeneous and formulated a media-independent specification of my replayable workspaces so as to reduce the impact such heterogeneity. I showed the characterization of replayable workspaces in terms of heterogeneous temporal relationships and introduced a formal model for the integration of these richly heterogeneous relationships.

The notions of media-independence presented in this chapter are coupled with the flexible architecture presented in Chapter 6 so as to preserve the transparency to media/tool characteristics of these heterogeneous replayable workspaces. My research on replayable workspaces provides a media-independent framework for the support of future transportable re-executable workspaces. I showed the modeling to be media/tool unaware so as to facilitate the specification of yet to be known workspaces. In the next chapter (Chapter 8), I describe key issues about our media-independent framework for integrating heterogeneous media on these replayable workspaces.

# CHAPTER 8

# THE STREAMING OF RE-EXECUTABLE CONTENT AND OTHER HETEROGENEOUS STREAMS

*"Do what you can, with what you have, where you are."*

— *Theodore Roosevelt.*

## 8.1 Introduction

The playback of re-executable content streams, together with continuous streams, arises during the streaming of inputs to multiple applets in a workspace (as examined earlier in Chapter 7). The streaming of re-executable content is also found in next generation internet downloadable content technologies such as with the program capsules of Tennenhouse's proposed active networks [102]. In general, the streaming of re-executable content arises whenever a remote repository is used to store state and input behavior of one or more distributed digital appliances, for example in internet computers and next generation wireless personal computing devices. As the quality of these digital consumer appliances increases, the need for policies to integrate richer and more complex forms of interactions will become evident.

Applet input streams (herein applet streams) contain re-executable events such as simple window events or even abstract operations. Because the re-execution of these events is asynchronous and stateful, playback mechanisms need to account for performance differences between record/replay workstation conditions. Furthermore, playback is complicated by the need to preserve different types of temporal relationships. For example, the playback of the notebook and graphing tools requires enforcing a causal relationship (such as for cut and paste) between them. On the other hand, the playback of audio-annotated pointer gesturing requires only a temporal (non-causal) relationship (Chapter 7, see Fig. 7.5).

Relationships between events across applet streams may be different. By taking advantage of (1) the tolerance of a relationship to asynchrony and (2) the tolerance of stream scheduling to discontinuities, it is possible to enforce multiple such relationships during playback. The heterogeneity of these streams and their inter-relationships affect the efficiency of streaming mechanisms. I present a framework for the integration of heterogeneous relationships between $n$ applet streams. The framework and its mechanisms are tool and media independent. The resulting media integration problem is approached as a constraint update propagation problem. Relationships between applet streams are translated into dependency constraints to be enforced. Enforcement is performed periodically. Periodical reports from an applet stream provide updates to the constraint fulfillment problem. Until all dependency constraints for a relationship are satisfied, a synchronization treatment measure is applied to a reporting applet stream. The treatment measure is determined at run-time based on the synchronization policy and playback tolerances being used.

In this chapter, I describe novel mechanisms for the integration of heterogeneous requirements. The framework provides multiple treatment measures to meet various inter-stream relationships requirements. In particular, I describe the use of statistical process controls (SPC/SQC) as the foundation of heuristics to introduce long-term guidance over the use and need for synchronization and control its associated costs over playback continuity. I describe mechanisms based on the relaxation of scheduling and/or synchronization effort. These adaptive mechanisms are controlled by a region of statistical tolerance over the run-time behavior of the inter-stream asynchrony between streams in a synchronization relationship.

## 8.2 Motivation and Requirements

Our motivation is to extend replay-awareness to computer-based workspaces. The requirements for the support of replayable workspaces were presented in Chapter 6. In Chapter 7, I examined the specification and modeling of replayable workspaces so as to support the media-independent integration of heterogeneous streams. In this chapter, we desire to formulate a playback media integration model that supports the heterogeneity introduced by re-executable content streams and it is yet flexible enough to incorporate new and *unknown* stream types to be found in future replayable workspaces.

## 8.3 Synchronization and Continuity

From our results in [68], both playback smoothness and continuity estimators depend on the synchronization costs $q$ as follows.

- playback smoothness of asynchronous streams;

  We propose to relate playback smoothness to how schedule time departs from elapsed real-time. For a smooth playback, we want such departures to be small and stable. Let $m_i$ be the ratio of schedule time (LTS) to real time (RT), i.e., $m_i = \frac{LTS_i - LTS_{i-1}}{RT_i - RT_{i-1}}$. Because we are dealing with re-executable content, scheduling needs to be adjusted to compensate for trends during playback on workstations of unequal performance. For example, such compensations are found on the compression and/or expansion of resources such as: (1) idle schedule time [52, 66], (2) silence [31, 45], and (3) presentation frame rate [76]. Let $w_i$ be the adaptation effort over a schedule time interval $T$, thus resulting in the compression (or expansion) of $T_i$ into a playback real-time duration $w_i T$. At the end of $T$, synchronization is performed with associated costs $q(a)$. Updating our metric $m_i$ with these values, we find that playback smoothness is affected by synchronization costs $q(a)$ as follows:

  $$m_i(a) = \frac{T}{w_i T + q(a)} \tag{8.1}$$

  The compensation effort $w_i$ reduces the impact of synchronization costs over playback smoothness. Therefore, we want, over time, a compensation effort that is smooth and effective. That is, we desire $m_i$ to exhibit exhibit controlled variability over an average value of 1.

- continuity jitter of continuous streams;

  Let $c_i$ be the timing departure from the expected duration ($c$) of a media frame to the actual real-time elapsed during its playback: $c_i = \frac{c}{RT_i - RT_{i-1}}$. At the end of the playback of the frame, synchronization is performed with associated costs $q(a_i)$. Updating our metric $c_i$ when $T = c$, we find that playback continuity is affected by synchronization costs as follows:

$$c_i(a) = \frac{T}{T + q(a)} \tag{8.2}$$

No surprise here. When synchronization costs are small, the metric $c_i$ tends to 1. When synchronization costs are large, $c_i$ is smaller than 1. When synchronization costs are variable (continuity jitter), $c_i$ exhibits strong variability over an expected value of 1. We desire the metric $c_i$ to exhibit exhibit controlled variability over an average value of 1.

Similarly, synchronization is easily related to synchronization costs, as follows. Synchronization costs $q(a)$ are dependent on the inter-stream asynchrony $\epsilon_i$. The synchronization costs $q(a)$ depend on the size of synchronization wait $w(a)$ imposed by the session manager during playback. Among other things, this waiting time $w(a)$ depends on the current inter-stream asynchrony $\epsilon_i$ and the treatment measure selected by the session manager to treat the asynchrony $\epsilon_i$.

**Discussion**

To improve playback continuity and smoothness, we desire low and stable synchronization costs. To achieve (1) low and (2) stable synchronization costs, we need (respectively) (1) efficient and (2) long-term controls over the inter-stream asynchrony. To obtain efficient and stable synchronization mechanisms, we desire to either (1) decrease synchronization costs and/or (2) decrease the frequency of synchronization checks. The first class of mechanisms is based on compensation mechanisms; the latter class is based on relaxation mechanisms. In this chapter, I present various such mechanisms and a framework for their integration.

## 8.4 The Streaming of Re-executable Content

Suppose that a session is composed of two streams $a$ and $b$. In turn, each stream is composed of $n$ scheduling intervals, as follows:

$$\left\{ \begin{array}{l} a = (T_{a1}, T_{a2}, \cdots, T_{an}); \\ b = (T_{b1}, T_{b2}, \cdots, T_{bn}); \end{array} \right\} \tag{8.3}$$

The playback of $(a, b)$ streams results in some observed playout schedules given by:

$$\left\{ \begin{array}{l} a* = (T^*_{a1}, T^*_{a2}, \cdots, T^*_{an}); \\ b* = (T^*_{b1}, T^*_{b2}, \cdots, T^*_{bn}); \end{array} \right\} \tag{8.4}$$

During playback, the media integration mechanism attempts to adapt playout schedules to meet the *relative timing constraint* (i.e., time equals):

$$\left\{ \begin{array}{l} T^*_{a1} \approx T^*_{b1}; \\ \cdots \\ T^*_{an} \approx T^*_{bn}; \end{array} \right\} \tag{8.5}$$

For continuous media streams, because of their periodicity, the duration of record and replay scheduling intervals is the same (i.e., $T_{ai} = T^*_{ai}$). Furthermore, because continuous media is relatively stateless, it is always possible to enforce, during record of $(a, b)$ streams, that the periodicity of both streams be the same, i.e., $T = T_{ai} = T_{bj}$. These characteristics facilitate meeting the (time-equals) relative timing constraint.

On the other hand, the streaming of re-executable content departs from these two characteristics, as follows. First, because event playout time is asynchronous and unpredictable, the duration of record and replay scheduling intervals is not likely to be the same (i.e., $T_{ai} = T^*_{ai} + f_o(a_i)$). Similarly, it is not likely either that the duration of scheduling intervals will be the same across both $(a, b)$ streams, all the time, (i.e., $T_{ai} \neq T_{bj}$). These characteristics complicate media integration of the (time-equals) relative timing constraint as follows:

$$\left\{ \begin{array}{l} T^*_{a1} + f_o(a_1) = T^*_{b1} + f_o(b_1); \\ \cdots \\ T^*_{an} + f_o(a_n) = T^*_{bn} + f_o(b_n); \end{array} \right\} \tag{8.6}$$

The approach in this dissertation relies on the use of per-stream scheduling. This is different from the use of global timing (that is, the reliance on absolute scheduling intervals timing across streams). Instead, the relative stream scheduling proposed in this dissertation addresses departures from scheduling faithfulness through the timing variability model presented in Chapter 4. I review these next.

### 8.4.1 Definition of Playback Asynchrony Conditions

There are two types of playback asynchrony conditions that characterize the playout times for re-executable content media on

- **random timing departures:**

  Random timing departures are due to varying load conditions during playback on a replay workstation of similar performance to the record workstation. These departures are represented by the asynchrony estimate:

$$\epsilon_i = f_o(b_i) - f_o(a_i) \tag{8.7}$$

  where $\epsilon_i$ is a random variable that represents the asynchrony between streams $a$ & $b$.

- **long term asynchrony trends:**

  Long-term asynchrony trends are timing departures due to substained performance differential $(\frac{\delta_{replay}}{\delta_{record}})$ between record and replay workstations. Assuming an additive model for the accumulation of this differential over time, the $\epsilon_i$ time series manifests the following long-term trend:

$$\epsilon_i = \epsilon_{i-1} + \frac{\delta_{replay}}{\delta_{record}}(f_o(a_{i-1}) - f_o(b_{i-1})) \tag{8.8}$$

### 8.4.2 The Handling of Pair-wise Asynchrony Trends

Media integration relies on per-stream compensation measures on pair-wise temporal relationships. During playback, the adaptive scheduling mechanism (see Chapters 4 & 7) attempts to adapt the original playout schedule of two re-executable content streams $(a : T_{a1}, \cdots, T_{an})$ and $(b : T_{b1}, \cdots, T_{bn})$ from

$$\left\{ \begin{array}{l} T_{a1} \approx T_{b1}; \\ \cdots \\ T_{an} \approx T_{bn}; \end{array} \right\} \tag{8.9}$$

into $(1 - \omega_1(a))$ compensated playback schedules, $(a : T^*_{a1}, \cdots, T^*_{an})$ and $(b : T^*_{b1}, \cdots, T^*_{bn})$, so as to meet their *relative* timing constraints (when subject to timing variability) as

$$\left\{ \begin{array}{l} (1 - \omega_1(a))T^*_{a1} \approx (1 - \omega_1(b))T^*_{b1}; \\ \cdots \\ (1 - \omega_n(a))T^*_{an} \approx (1 - \omega_n(b))T^*_{bn}; \end{array} \right\} \tag{8.10}$$

Let's define the (relative) compensation effort as $c_\omega(i) = \frac{1-\omega_i(a)}{1-\omega_i(b)}$. Then, by Eq. 8.6, we can rewrite Eq. 8.10 into

$$\left\{ \begin{array}{l} c_\omega(1)T^*_{a1} - T^*_{b1} = f_o(b_1) - c_\omega(1)f_o(a_1); \\ \dots \\ c_\omega(n)T^*_{an} - T^*_{bn} = f_o(b_n) - c_\omega(n)f_o(a_n); \end{array} \right\} \tag{8.11}$$

When compared with Eq. 8.7, we can rewrite Eq. 8.11 into

$$\left\{ \begin{array}{l} c_\omega(1)T^*_{a1} - T^*_{b1} = \epsilon^*_1; \\ \dots \\ c_\omega(n)T^*_{an} - T^*_{bn} = \epsilon^*_n; \end{array} \right\} \tag{8.12}$$

where $\epsilon^*_i = f_o(b_i) - c_\omega(i)f_o(a_i)$ represents the *compensated timing departure.*

### Discussion

This is important, because now, the replay of pair-wised scheduling intervals from heterogeneous streams can be $\omega$-compensated to account for departures from relative timing faithfulness. The result, is a compensated timing departure, estimated by $\epsilon^*_i$. The compensation $c_\omega(i)$ introduces time deformations that scales up or down, as needed, the *relative* schedule of any re-executable content stream exhibiting an asynchrony condition. The compensation effort $c_\omega(i)$ is variable and adjusted on every scheduling interval.

Clearly, the convergence or stability of $c_\omega(i)$ is important. The stability of $c_\omega(i)$ is monitored through run-time statistical process controls over the *asynchrony* $\epsilon^*_i$ time series. When the *asynchrony* $\epsilon^*_i$ follows a long-term trend, SPC-based monitoring is used to detect the trend and adaptive scheduling discussed in Chapter 4 is used to remove the trend. The mechanism is somewhat more forgiving of short-term timing departures. This tolerance is an input parameter that specifies a region of statistical tolerance over the *asynchrony* $\epsilon^*_i$.

If $a$ and $b$ are re-executable content streams, then the relative compensation effort remains as $c_\omega(i) = \frac{1-\omega_i(a)}{1-\omega_i(b)}$. As discussed in Chapter 7, in practice it might be preferrable to adapt only one stream (say $a$) instead of adapting both — as this removes the presence of the convergence problem on $c_\omega(i)$. If $b$ is a continuous streams, then we have two choices. On one hand, we can model $b$ as a re-executable content stream so as to allow our statistical process controls to adapt the playback of $b$. On the other hand, we may want to model $b$ as a continuous stream. If $b$ is a continuous stream, then its associated compensation is negligible and the relative compensation effort reduces to $c_\omega(i) \approx (1 - \omega_i(a))$ — this is consistent with model presented in Chapter 4. If both $a$ and $b$ are continuous streams, then individual compensations are negligible and the relative compensation effort reduces to $c_\omega(i) \approx 1$. This later one is the inherent model assumed by existing continuous media servers (which lack our notions of long-term adaptive coupling).

## 8.5 Framework

The implementation of $n$-ary dependency constraints between multiple streams relies on a deferred synchronization handshake handled at the session manager using two media-independent messages: Update() and Continue(). Update() is a scheduling report used by an applet's temporal controller to report the completion of the current scheduling interval. Whenever an applet's temporal controller schedules a synchronization event, it performs the following tasks: (1) flushes and waits for the completion of buffered events, (2) suspends further scheduling, and (3) sends an Update() report to the session manager. The Continue() command is used by the session manager to allow an applet's temporal controller to resume scheduling of its next scheduling interval. The handshake has a net effect of *"pause-and-continue"* over the scheduling of individual applet streams. Note that while scheduling is paused at the applet, tasks such as prefetching can take place at an applet.

### 8.5.1 Applet Perspective: Pause and Continue

All synchronization policies rely on the synchronization handshake. The implementation of the various treatments is encapsulated within the session manager and is transparent to applets or even their stream controllers. Outside the session manager, treatment measures could be differentiated only by the ordering of messages to-and-from the session manager. For example, Fig. 8.1 shows the message ordering that occurs during the handling of a 2-way dependecy constraint ($rhs$ : $lhs$). Note that the reporting $rhs$ slave stream is blocked until a report is received from its $lhs$ master. Similarly, Fig. 8.2 shows the message ordering that occurs during the handling of a 0-way dependency constraint (i.e., streams are unrelated). Note that the reporting $rhs$ stream is not blocked and continues independently (and regardless the progress or lack of) from any other stream such as for example, $lhs$.

Figure 8.2: Use of the synchronization handshake to implement the non-blocking treatment measure. Since $a$ and $b$ are not related, when either one reports to the session manager, neither $a$ nor $b$ is force to wait for the other.

### 8.5.2 Session Manager Perspective: Temporal Constraint Resolution

At the session manager, the resolution of multiple dependency constraints between applet streams is represented as a constraint update propagation problem. As discussed in Chapter 7, each Update() report triggers an iteration over the session manager's constraint resolution algorithm. After satisfaction of the dependency constraints associated with the reporting stream, the session manager then releases the stream with a Continue() command. The satisfaction function is implemented through run-time process controls.

The constraint update propagation problem is setup as follows.

- STATE:
  a vector of $n$ elements that tracks the scheduling progress for each stream in terms of the last synchronization event seen so far at each one. The STATE vector is initialized to -1.

- TARGET:
  a vector of $n$ elements that tracks the satisfaction goal for each of stream in terms of the next synchronization event to see. The TARGET vector is initialized to zeroes.

- CONSTRAINT:
  the $W_C$ matrix introduced in Chapter 7. It is just the transpose of the tolerance matrix $W_T$.

- SATISFACTION(j):
  the $j^{th}$ row of the $W_C$ matrix used as an $n$-ary dependency specification tuple. This tuple generates dependency constraints as $d(j \to i)$ *such that* $(0 \leq c_{ji} < \infty)$. The satisfaction of stream $j$ is met when the pair-wise "*meet*" constraints associated to these dependency constraints are satisfied.

- MEET(j,i):
  pair-wise "*meet*" constraints are defined in Chapter 7, Eq. 7.14.

- UPDATE(j):
  causes the STATE(j) to be incremented and triggers the re-evaluation of the SATISFACTION function for every stream constrained by $j$.

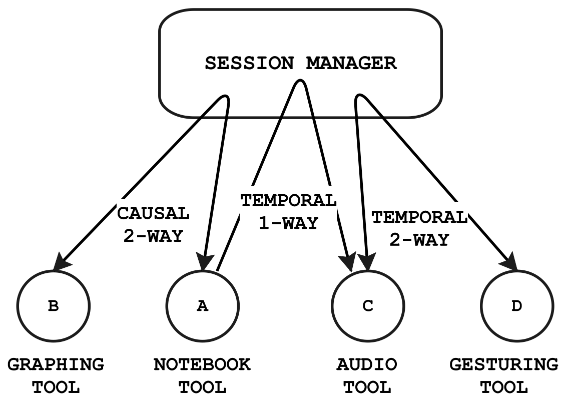


**Figure 8.3: The session manager handles the resolution of $n$-ary relationships. Relationships between applet streams are reduced into dependency constraints. A dependency constraint $c_{ba}(b \to a)$ is satisfied when streams $a$ and $b$ *"meet"*. Different policies specify different ways of "*meeting*". Stream $a$ is forced to wait by the session manager until its dependency constraints are satisfied.**

- RELEASE(j):

  causes TARGET(j) to be incremented and then triggers CONTINUE() command to $j$.

During the design of a workspace (see Fig. 8.3), the workspace designer specifies the tolerance matrix $W_T$ for the workspace (see Chapter 7, Eq. 7.7). The CONSTRAINT matrix follows then from Eq. 7.13. The session manager enforces the resulting $nxn$ dependency constraints $c_{ij} \in W_C$ during the playback of $W$. A dependency constraint $c_{ba}(b \to a)$ is satisfied when streams $a$ and $b$ *"meet"*. Different policies specify different ways for $a$ & $b$ to "*meet*". Stream $a$ is forced to wait by the session manager until its dependency constraints are satisfied. The session manager enforces a relative stream progress that satisfies (at each synchronization event) the $nxn$ dependency constraints that specify a workspace $W$.

### 8.5.3 Media Integration Perspective: Synchronization/Continuity

As stated, continuity or smoothness on the playback of an stream is affected by synchronizing costs (see also **Appendix C**). During playback, synchronization costs introduce a *"pause-and-continue"* discontinuity (for example, see Fig. 7.6 *wrt* Fig. 7.7 in Chapter 7). The synchronization cost $q(a_i)$ observed by an applet stream $a_i$ has two components: (1) the roundtrip time ($2f$) from $a_i$ (to and back the session manager) and (2) the waiting time $w(a_i)$ associated to the synchronization treatment imposed over $a_i$ by the session manager. Equivalently,

$$q(a_i) = 2f + w(a_i) \tag{8.13}$$

Synchronization costs $q(a_i)$ are dependent on the inter-stream asynchrony $\epsilon_i$. The synchronization costs $q(a_i)$ depend on the synchronization wait $w(a_i)$ imposed during playback by the session manager. Among other things, this waiting time $w(a_i)$ depends on the current inter-stream asynchrony $\epsilon_i$ and the treatment measure selected by the session manager to treat the asynchrony $\epsilon_i$. By carefully allocating treatment effort over $\epsilon_i$, we can control the efficiency of synchronization costs $q(a_i)$. Similarly, by engineering long-term process controls, we can stabilize synchronization costs and consequently maintain the smoothness or continuity of the playback. As a result, synchronization to continuity tradeoffs are specifiable.

There are three possible asynchrony conditions between two applet streams $a$ and $b$: (1) $a$ is ahead of its $b$ ($R_L$); (2) $a$ is behind of its $b$ ($R_R$); and (3) $a$ is in-sync to its $b$ ($R_C$). The central region $R_C$ represents an asynchrony tolerance region for which $a$ and $b$ are considered "relatively" synchronized (see Fig. 8.4). In this region, $\epsilon_i$ is regarded as non-significant. Synchronization is triggered only when $\epsilon_i$ exceeds this tolerance range. By varying the specification $(\sigma_L, \sigma_R)$ of this region, we can indirectly control the synchronization effort imposed over an applet stream $a_i$.

The general specification of a workspace $W = \{W_A, W_R, W_T\}$ (Fig. 7.5) is of the form:

$$W = \left\{ \begin{array}{l} W_A = (A, B, C, D); \\ W_R = (R_1(A : B), R_2(C \leftarrow A), R_3(C : D)); \\ W_T = ([\sigma = 0], [\sigma = Q_{R2}], [\sigma = Q_{R3}]); \end{array} \right\} \tag{8.14}$$

where $Q_{R_k}$ is the specification of the tolerance region for the relationship $R_k()$. An instance of this workspace specification (for this particular example) was given in Chapter 7, equation 7.12. When $Q_{R_k}$ equals 0, the tolerance region is nil; synchronization is never relaxed. This suits causal relationships. For temporal relationships, the relaxation of synchronization is controlled by the $Q_{R_k}$ specification. A similar tuple specification can be formulated for the relaxation of scheduling.

To describe the synchronization policies of the framework, we need to specify how is scheduling done and how is synchronization done? We describe these for various policies for the handling of the relationship $R_k(a \leftarrow b)$ between applet streams $a$ and $b$.

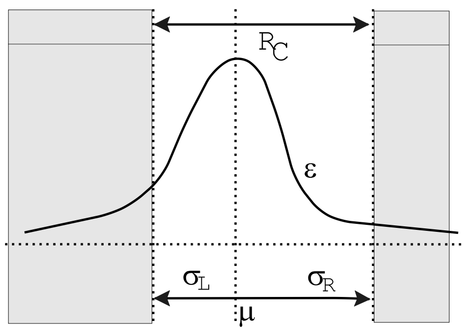


**Figure 8.4: The region $R_C$ represents an asynchrony tolerance region for which $a$ and $b$ are considered "relatively" synchronized. In this region, $\epsilon_i$ is regarded as non-significant. Synchronization measures are fired only when $\epsilon_i$ falls outside $R_C$. By varying the specification $Q_{Rk} = (\sigma_L, \sigma_R)$ of this region, we control the synchronization costs $w(a_i)$ over the scheduling of streams in the relationship $R_k$.**

### P1 Policy

The P1 policy relies on short-term scheduling corrections. If, during playback, a scheduling interval last less than during record, an idle time adjustment is inserted to match the current timing departure. If the scheduling interval last more, nothing is done. There is no synchronization between $a$ and $b$.

### P2 Policy

The P2 policy relies on short-term scheduling corrections. Synchronization is 1-way between $a$ and $b$; whenever $b$ is ahead of $a$, $b$ waits for $a$.

### P3 Policy

The P3 policy relies on long-term scheduling corrections. Synchronization is 1-way; whenever $b$ is ahead of $a$, applet stream $b$ waits for $a$. The long-term scheduling compensation is computed during run-time, based on the statistical process performance of the relationship (see Fig. 8.5). Our scheduling controls implement long-term measures over the scheduling of a stream as follows. For a given relationship, we specify a tolerance region $(\mu_\epsilon, \sigma_\epsilon)$ over a smoothed indicator of the inter-stream asynchrony $\epsilon_i$. The tolerance region is used to characterize the compensation effort *wrt* the asynchrony departure. The

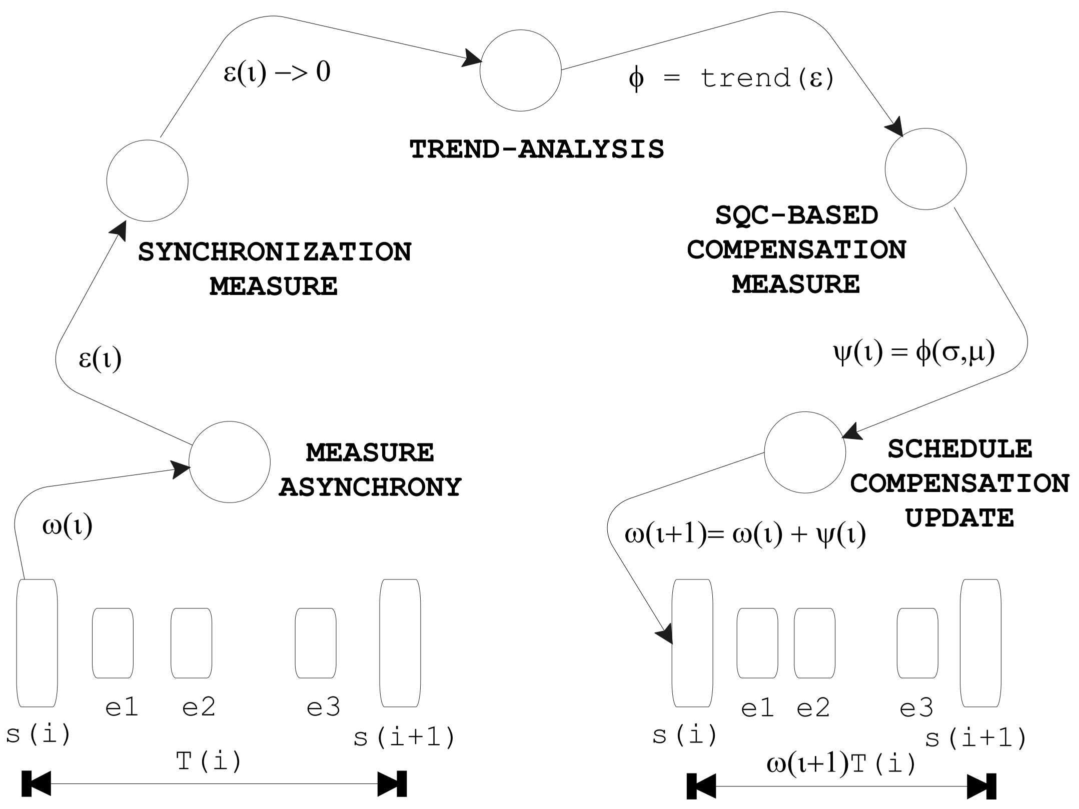


**Figure 8.5: Process control model for the long-term adaptive scheduling controls. For a given relationship, we specify a tolerance region $(\mu_\epsilon, \sigma_\epsilon)$ over a smoothed indicator of the asynchrony $\epsilon_i$. A statistical tolerance region characterizes the compensation effort *wrt* to $\epsilon_i$. The compensation strength is determined by the statistical significance of the smoothed asynchrony departure.**

strength of the compensation effort is related to the statistical significance of the smoothed asynchrony departure.

### P4 Policy

The P4 policy relies on short-term scheduling corrections. Synchronization is 2-way; whenever $b$ is ahead of $a$, $b$ waits for $a$ and whenever $a$ is ahead of $b$, $a$ waits for $b$.

### P5 Policy

The P5 policy relies on long-term scheduling corrections. Synchronization is relaxed 2-way. Whenever $b$ is ahead of $a$, $b$ may wait for $a$ and whenever $a$ is ahead of $b$, $a$ may wait for $b$. This relaxation of synchronization is based on the run-time statistical process performance of the relationship. The relaxation of synchronization firing controls is implemented as follows (see Fig. 8.6). For a given relationship, we specify a tolerance region $(\mu_\epsilon, \sigma_\epsilon)$ over a smoothed indicator of the inter-stream asynchrony $\epsilon_i$. The tolerance region is used to characterize the relaxation effort *wrt* the statistical significance of the asynchrony departure. Blocking synchronization measures are initiated when a statistical significant trend and/or departure is detected. Adaptive scheduling measures (as specifed by the P3 policy) may be coupled with this mechanism to compensate for long-term playback performance trends.

## 8.6 Performance of Heterogeneous Media Integration

In this chapter, I provide preliminary analysis of the performance of heterogeneous media integration under timing variability. Our goal here is to simulate timing variability over the playback of streams and determine the effectiveness of media integration for heterogeneous media streams.

Our playback model assumes that the playback of an applet stream is subject to timing departures $f_o()$. The playout duration $t^*(e_i)$ of an event $e_i$ is modeled as subject to a timing departure $f_o(e_i)$ over the event's original playout duration $t(e_i)$, that is, $t^*(e_i) = t(e_i) + f_o(e_i)$. However, the mechanisms and policies presented in this dissertation operate at an interval grain. Consequently, we analyze timing variability at the scheduling interval

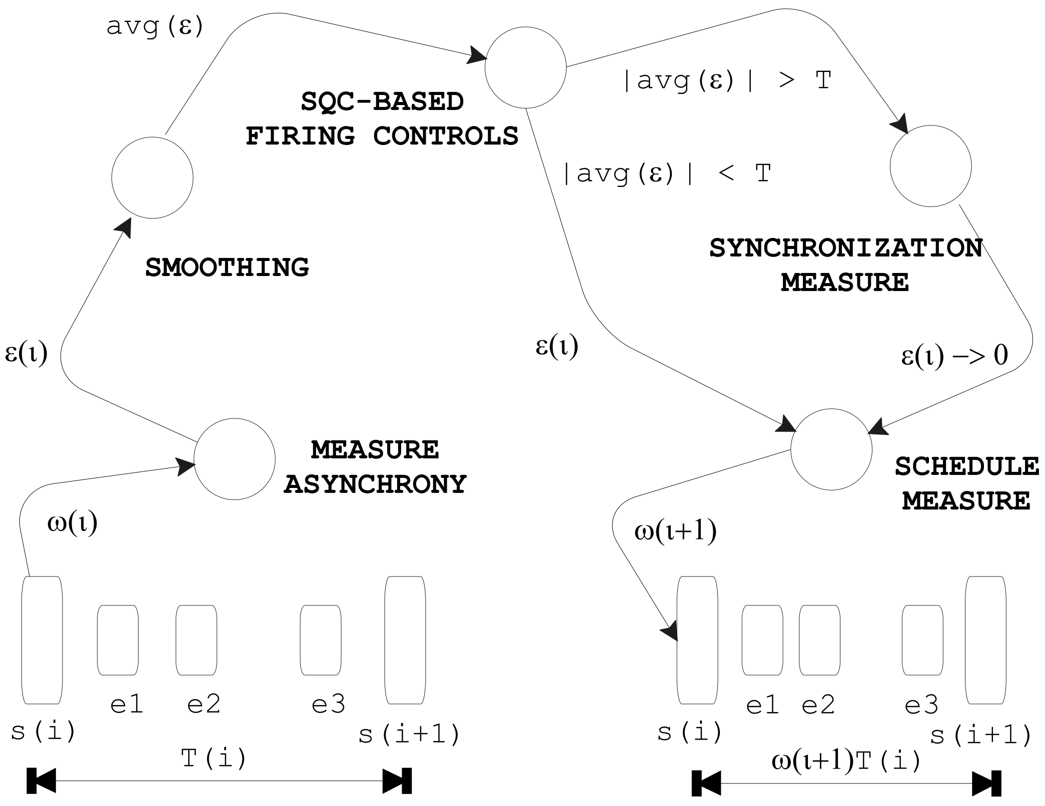


**Figure 8.6: Process control model for relaxing synchronization firing controls. For a given relationship, we specify a tolerance region $(\mu_\epsilon, \sigma_\epsilon)$ over a smoothed indicator of the asynchrony $\epsilon_i$. A statistical tolerance region characterizes the relaxation effort *wrt* to the asynchrony departure. Blocking synchronization measures are initiated when a statistical significant trend and/or departure is detected.**

grain, as follows

$$T_i^*(a) = T_i(a) + f_o(T_i(a)) \tag{8.15}$$

To study the effect of timing variability over the playback of heterogeneous streams, we model the timing variability component $f_o()$ as a random variable with an uniform distribution. To characterize $f_o()$ we need to specify just two parameters: (1) the mean of the timing departure ($\mu_{f_o}$) and (2) the timing variability component ($\sigma_{f_o}$) to be imposed over the playback schedule. Equivalently,

$$T_i^*(a) = T_i(a) + \underbrace{\mu_{f_o(a)} + \sigma_{f_o(a)}}_{f_o(T_i(a))} \tag{8.16}$$

### 8.6.1 Design of the Experiments

When comparing <u>long-term asynchrony playback trends</u> (see Chapter 8, Section 8.4.1) between record and replay schedules $T_i$ and $T_i^*$, respectively, we need to consider three basic scenarios.

FAST: $T_i^*$ tends to execute faster than $T_i$.

SLOW: $T_i^*$ tends to execute slower than $T_i$.

BASE: $T_i^*$ tends to execute at a similar rate as $T_i$.

On the other hand, when analyzing <u>short-term timing departures</u> (see Chapter 8, Section 8.4.1) over the replay schedule $T_i^*$, we need to consider an additional scenario.

SAME: is like FAST but under stronger random perturbations over $f_o()$ (i.e., timing variability). This experiment is designed to test the interaction effect between both (short and long term) playback asynchrony conditions

Table 8.1 specifies the selected replay schedules used to design such experiments. These replay schedules correspond to the following timing departure models. The FAST is associated to the statistical timing departure of $f_o() : [\mu = \frac{-T}{4}; \sigma_T]$. The SLOW is associated to the statistical timing departure of $f_o() : [\mu = \frac{T}{4}; \sigma_T]$. The BASE is associated to the statistical timing departure of $f_o() : [\mu = 0; \sigma_T]$. The SAME is associated to the statistical timing departure of $f_o() : [\mu = \frac{-T}{4}; 2\sigma_T]$.

Fig. 8.7 shows the behavior of these schedule departures by plotting the timing variability component $t^*() - t()$ between playback and record schedules for these four basic

| LOAD | MODEL |
|---|---|
| FAST | $T^* = \frac{3}{4}T + \sigma_T$ |
| BASE | $T^* = 1T + \sigma_T$ |
| SLOW | $T^* = \frac{5}{4}T + \sigma_T$ |
| SAME | $T^* = \frac{3}{4}T + 2\sigma_T$ |

**Table 8.1: Simulation models used in the experimental design for testing the timing variability effect between record and replay.**

scenarios. The BASE schedule departure introduces a steady differential between record and replay, on which replay executes slightly slower (about $50ms$ per scheduling check). This results in a long-term asynchrony trend. The FAST schedule departure introduces a linear differential between record and replay, on which replay executes slightly faster (about 15% faster per scheduling check). This also results in a long-term asynchrony trend. The SAME schedule departure resembles the FAST (the replay executes at about 15% faster per scheduling check, too). However, it introduces larger random perturbations combined with a long-term asynchrony trend. These perturbations add up with the long-term trend resulting on an even larger long-term asynchrony trend than the one observed by FAST. Last, the SLOW schedule departure introduces a strong differential between record and replay, on which replay executes slower (about 30% slower per scheduling check). This results in a long-term asynchrony trend.

### 8.6.2 Baselining of the Architecture

This section baselines the performance of the architecture so as to provide a perspective to the subsequent analysis. Here, I analyzed three questions. First, are the inter-process communication overheads too large? Second, are the underlying timing services reliable? Last, how tight is the synchronization provided by the architecture? The answers are next.

Synchronization handshakes (see Section 8.5.1) rely on inter-process communication (IPC) services. We assumed the IPC round-trip time cost to be negligible. We will like to verify this assumption. To determine IPC overheads, we measured IPC round-trip time for a typical handshake exchange. The size of each messages was exactly 50 bytes. We sampled $n = 1000$ such round-trips. Individual samples were randomly time-spaced to

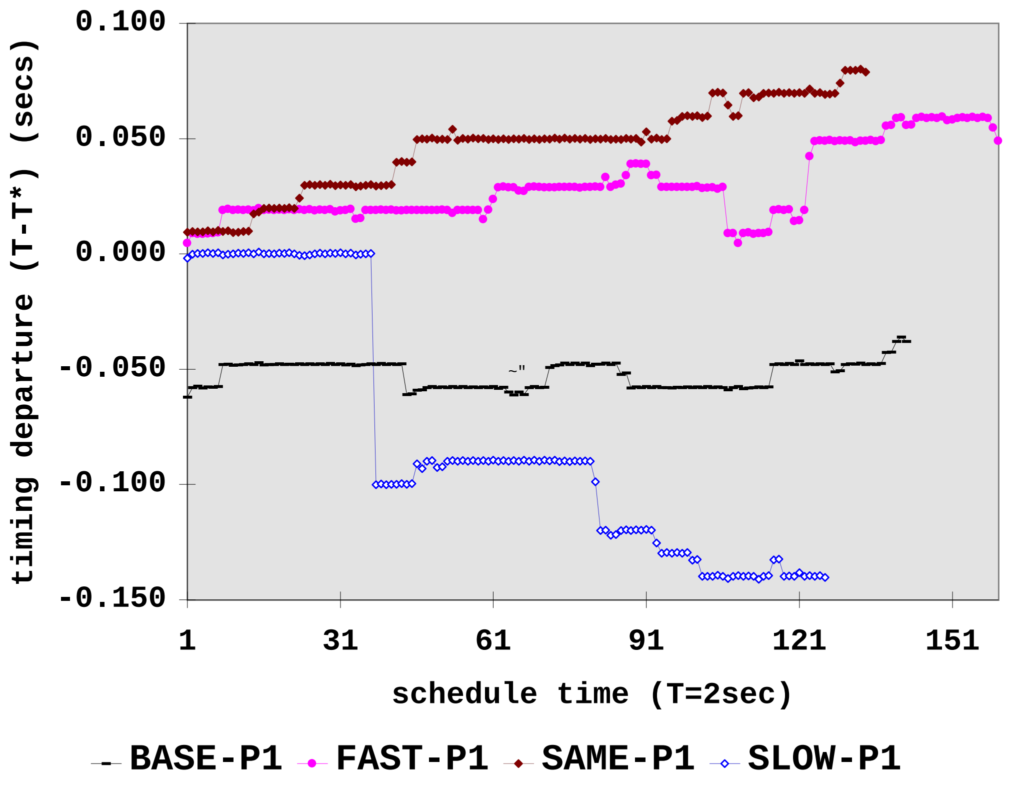


Figure 8.7: Schedule departures introduced by the timing variability models of BASE, FAST, SAME, and SLOW. The schedule departure is defined as the difference $t^*() - t()$ between playback and record schedules.

avoid IPC saturation and their side-effects over sequentiality (e.g., caching). A trial mean was extracted from the $n = 1000$ samples. The trial was repeated a total of $m = 10$ times and a grand-mean for these trial means was extracted. In the average, *intra-workstation* IPC round-trips take about

$$2f = IPC_\mu \pm IPC_\sigma \approx 2.2ms \pm 1.1ms|_{nm=10(1000)} \tag{8.17}$$

IPC overhead is at least an order of magnitude smaller than typical synchronization costs. IPC round-trip overhead is negligible for workstation-centric workspaces.

Scheduling accuracy depends on the reliability of the underlying timing services. To examine the reliability of timers, we analyzed timing behavior for the domain requirements of session playback, typically from $10ms$ to $1000ms$. The same test was performed under various load conditions. To avoid side-effects of sequential sampling, random pauses were inserted between consecutive sampling of the population. Timing services exhibited a statistically-biased behavior to exceed by 10% the requested service times. Equivalently,

$$\hat{t}_{sleep} = t_\mu \pm t_\sigma \approx t + .10t|_{10 \leq t \leq 1000ms} \tag{8.18}$$

Unless compensated during playback, this positive bias induces a long-term playback asynchrony trend over the playback of all streams. A differential scheduler such as the one presented in Chapter 4 is one mechanism to reduce the impact of timing services inaccuracy over scheduling.

Each stream is modeled as a sequence of scheduling intervals $T_1, T_2, \cdots, T_n$. The duration of the scheduling intervals $T_i$ need not be the same, but without loss of generality, we will assume that they are with regard to the following question. Is there an optimal scheduling interval for all and every media stream type? Intuitively, synchronization at too short intervals results is inefficient as processing overheads dominate processing. On the other hand, too long an interval between synchronization, results in relatively large asynchrony due to the unfrequent synchronization effort and timing variability. The next baseline experiment tested the relationship between scheduling interval and the inter-stream asynchrony. The playback of two streams (of negligible timing variability) was simulated under the BASE asynchrony condition under the P4 policy. The simulation was repeated on a total of $(n = 10)$ trials where each trial consisted of 300 scheduling intervals. Each trial tested the scheduling interval duration $(1 \leq T \leq 4)$ and observed its effect over the inter-stream asynchrony. The mean of the asynchrony as well as its standard deviation exhibited

| $T$ | $\mu_{async(in)}$ | $\sigma_{async(in)}$ |
|---|---|---|
| $1.0s$ | $1ms$ | $15ms$ |
| $2.0s$ | $4ms$ | $28ms$ |
| $3.0s$ | $7ms$ | $45ms$ |
| $4.0s$ | $5ms$ | $30ms$ |

**Table 8.2: Relationship between asynchrony and scheduling interval ($n =$ 10 trials).**

a linear dependency on $T$ for the range of 1 through 3 seconds, summarized below.

$$\hat{\sigma}_{async} = \frac{14.6T + 3}{1000}|_{R^2=0.9954, nm=10(300)} \tag{8.19}$$

$$\hat{\mu}_{async} = \frac{2.7T - 1.6}{1000}|_{R^2=0.9988, nm=10(300)} \tag{8.20}$$

Interestingly, both indicators also exhibited a drop at $4s$. The results are summarized in Table 8.2.

### 8.6.3 Comparative Performance

Experiments were designed to understand the performance of our adaptive policies for heterogeneous media integration by studying the limitations of policies P1, P2, and P4 under a range of load conditions. In particular, we were interested on studying the performance of these policies over media-oriented QoS. In Section 8.3, we constructed two such quality indicators for comparing the performance of heterogeneous streams in our architecture: inter-stream asynchrony and intra-stream continuity.

To determine the effects of long-term asynchrony playback trends over the non-adaptive protocols (P1, P2, and P4), the following experiment was designed. The playback of a session object was simulated. The session object was composed of two streams, referred to as $A$ (for audio) and $D$ (for re-executable data inputs). Such model represents the audio-annotated playback of an applet. The playback of such session requires strong playback continuity for $A$ and tight synchronization between $A$ and $D$. From our previous baseline experiments, (found in Chapter 4), a reliable characterization of $A$ was found to be $\mu(A) = T = 2s$ with an associated $\sigma_T(A) = 50ms$. In the experiments described below, such model is used to simulate $A$ as a continuous stream.

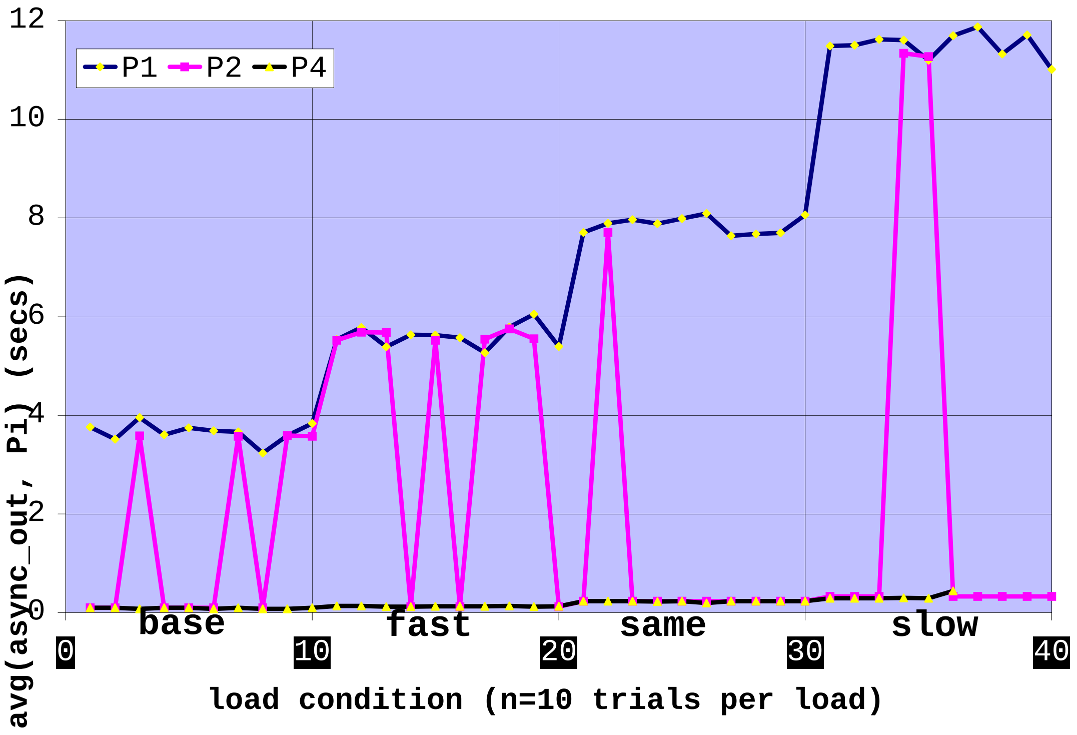


Figure 8.8: Effectiveness of P1 (0-way), P2 (1-way), and P4 (2-way), in reducing asynchrony during playback of $A$ and $D$ (a re-executable content stream). Under no long-term asynchrony trends (BASE), policies perform satisfactorily. Under long-term asynchrony trends, only P4 does. For each load condition, 10 trials were ran; each trial had at least 150 scheduling intervals. A total of 40 trials were ran. For each load condition, the average asynchony $\epsilon^*_{ab}$ of each of the 10 trials is plotted. Protocol P2 swings between low and high because it is a 1-way (asymmetrical) protocol.

Non-adaptive protocols P1 (0-way policy coupled with timing checks), P2 (1-way policy coupled with timing checks), and P4 (2-way policy coupled with timing checks) were compared in terms of their effectiveness in reducing asynchrony during the playback of $A$ and $D$. The corresponding $W_T$ tolerance matrices were:

$$W_T(P1) = \left\{ \begin{array}{cc} -1 & \infty \\ \infty & -1 \end{array} \right\}; \; W_T(P2) = \left\{ \begin{array}{cc} -1 & \infty \\ 0 & -1 \end{array} \right\}; \; W_T(P4) = \left\{ \begin{array}{cc} -1 & 0 \\ 0 & -1 \end{array} \right\} \quad (8.21)$$

The experiments were conducted under controlled playback asynchrony trends (BASE, FAST, SAME, and SLOW — baselined in the previous section) applied over the $D$ re-executable content stream. The experiment was repeated a total of $n = 10$ trials. Each trial consisted of at least 150 scheduling intervals. The results are shown in Fig. 8.8. As reported in Chapter 4, under the BASE region, each protocol performed satisfactorily. However, under the presence of strong playback trends, only the 2-way protocol P4 performs satisfactorily. Protocols P1 and P2 were unable to guarantee asynchrony *wrt* variation on playback asynchrony trends on as the load the synchronization hold. Our next goal was to gain some insight into the performance characterization of the P4 policy and understand its impact over media-oriented QoS indicators so as to support the integration of heterogeneous media.

### 8.6.4 Characterization of the P4 Policy

Protocol P4 is effective in reducing the inter-stream asynchrony during the playback of $A$ and $D$. In average, each synchronization check cuts the incoming asynchrony in half (i.e., $\mu_{async(in)} \approx 160ms$ into $\mu_{async(out)} \approx 80ms$). These results are found in Fig. 8.10. The same setup was repeated with four streams and similar results were found. It appears that $\mu_{async(out)} = \frac{\mu_{async(in)}}{2}$. Protocol P4 is also effective in preserving the intra-stream continuity of $A$ and $D$ streams. Each synchronization check relies on the use of idle waiting time (*waittime*) to re-synchronize the streams at the session manager. The waittime introduced by each synchronization check introduces two levels of discontinuity: (1) one of negligible impact (over A) $\mu_{waittime} < 10ms$. (2) one of substantial impact (over D) $\mu_{waittime} > 100ms$. These results are shown in Fig. 8.10.

This performance characterization is not user-specifiable. Insight on how to specify such performance tradeoffs over policy P4 should allow us to accommodate a range of heterogeneous media requirements.

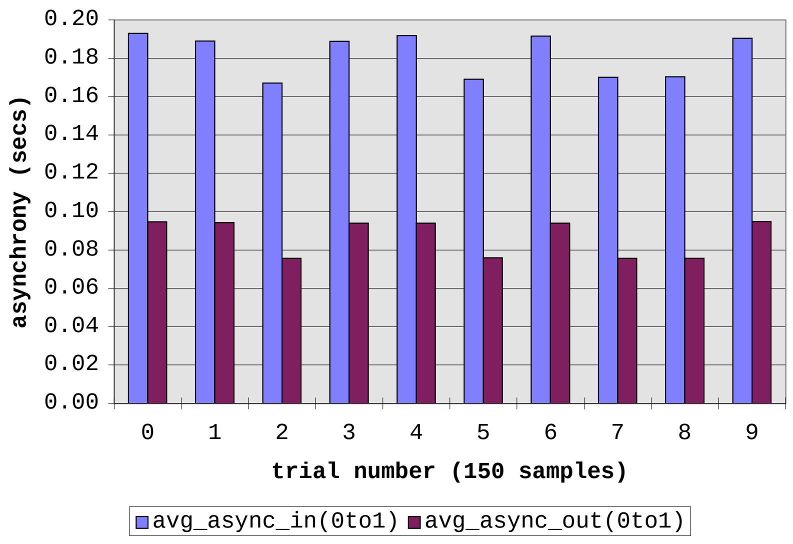


**Figure 8.9: Effectiveness of P4 in reducing asynchrony during the playback of a continuous and a re-executable content stream. In average, each synchronization check cuts the incoming asynchrony in half (i.e., $\mu_{async(in)} \approx 160ms$ into $\mu_{async(out)} \approx 80ms$).**

### 8.6.5 Relaxation of the P4 Policy

Fig. 8.11 gives insight into the performance of protocol P4 *wrt* the incoming inter-stream asynchrony $async(in)$ during the playback of $A$ and $D$. Typically, protocols based on the P4 policy react to either the current asynchrony or some short-term smoothed indicator of it. The incoming asynchrony is characterized by parameters: ($\mu_{async(in)} \approx 0.650ms$, $\sigma_{async(in)} \approx 0.650ms$). The resulting waittime is characterized by parameters: ($\mu_{waittime} \approx 0.350ms$, $\sigma_{waittime} \approx 0.650ms$). Similarly, the outgoing asynchrony is characterized by parameters: ($\mu_{async(in)} \approx 0.250ms$, $\sigma_{async(in)} \approx 0.650ms$). Again, although on the average, each synchronization check cuts the inter-stream asynchrony in half, because of the lack of effective measures over underlying long-term asynchrony playback trends, the incoming asynchrony is never removed. Both our media-oriented QoS indicators — $asynch_{in/out}$ and $waittime$ — lack a *state of statistical process control.* Furthermore, the performance of protocol P4 is not user-specifiable.

Beyond that naive look, Fig. 8.11 gives important insight into the relaxation of protocol P4 proposed by adaptive protocol P5. Previously, in Fig. 4.10 of Chapter 4, we showed

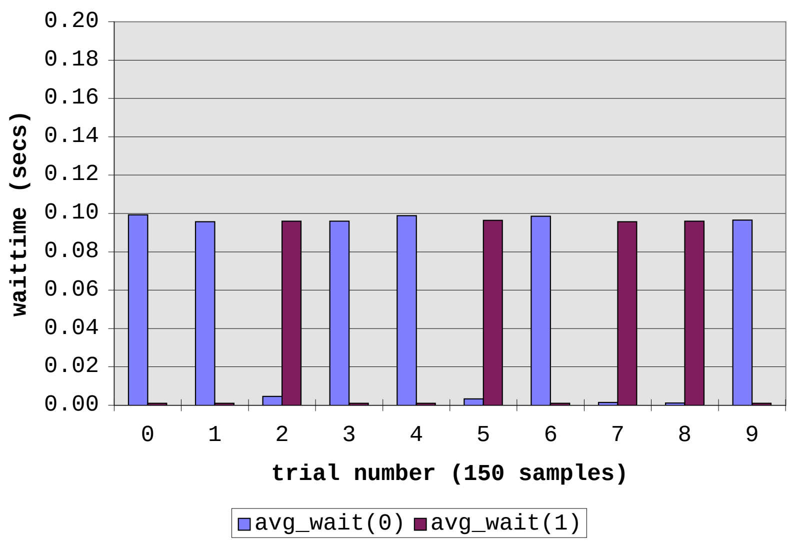


**Figure 8.10: Effectiveness of P4 in preserving playback continuity. The wait time introduced by each synchronization check introduces two levels of discontinuity: (1) one of negligible impact $\mu_{waittime} < 10ms$. (2) one of substantial impact $\mu_{waittime} > 100ms$.**

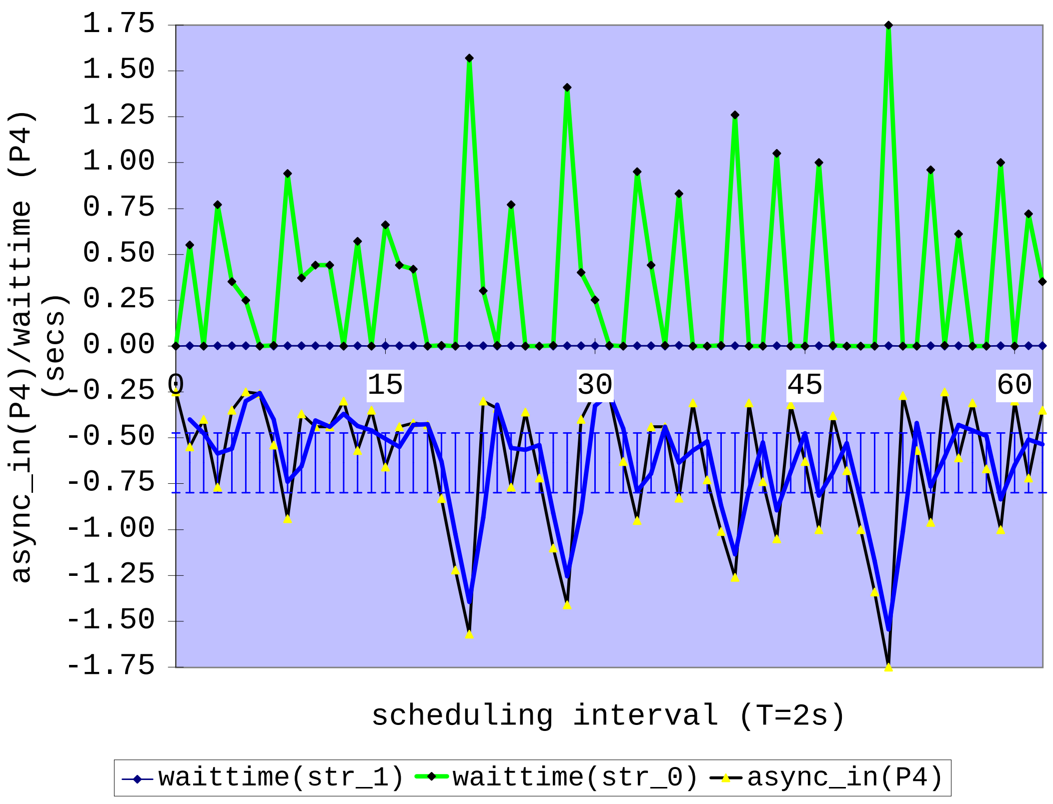


**Figure 8.11:** **Insight into the relaxation of protocol P4 proposed by adaptive protocol P5. Protocol P5 first smooths the asynchrony and imposes a tolerance region ($0.50 \leq \hat{async}_{in} \leq 0.75$) over the smoothed asynchrony (bottom curve). When the smoothed asynchrony indicator exceeds the tolerance region, the underlying 2-way synchronization controls of protocol P4 are used. Such use of synchronization barriers introduces a discontinuity on the playback of slave streams (top curve). Under protocol P5, when the asynchrony indicator is under statistical control, synchronization is relaxed. As a consequence, the playback discontinuity that normally will be observed by protocol P4 is removed. Online statistical process controls are used to monitor the performance of the smoothed asynchrony indicator. Protocol P5 allows user-specification of the process control over asynchrony and continuity between two streams.**

adaptive scheduling measures so as to remove the presence of long-term asynchrony trends during playback. The strength of the compensation effort was related to the statistical significance of the smoothed asynchrony departure (see **Appendix D**). Fig. 8.11 proposes a similar technique (see **Appendix E**): the relaxation of synchronization on the P4 policy as proposed by the P5 policy. Fig. 8.11 shows the asynchrony and its relationship with the imposed synchronization wait (*waittime*). Synchronization wait results in discontinuity during the playback. Protocol P5 works approaches flexible specification of tradeoffs between synchronization and continuity as follows. First, it smooths the asynchrony estimate and imposes a user-specifiable tolerance region over the smoothed indicator of the asynchrony. When the smoothed asynchrony indicator exceeds the tolerance region, the underlying 2-way synchronization controls of protocol P4 are used. Such use of synchronization barriers introduces a discontinuity on the playback of slave streams. For example, the strong discontinuities observed by protocol P4 are associated to the use of a nil tolerance region over the asynchrony. Under protocol P5, when the asynchrony indicator is under statistical control, synchronization is relaxed (i.e., no synchronization is performed). As a result, a playback discontinuity normally observed by protocol P4 is removed. Online statistical process controls are used to monitor the performance of the smoothed asynchrony indicator. Protocol P5 allows user-specification of the process control over asynchrony and continuity between two streams.

The relaxation or tightening of synchronization measures is specifiable with policy P5 by expanding or reducing (respectively) the tolerance region of a temporal relationship. The tolerance region is specified by two parameters: `USL` and `LSL` which stand for upper specification limit and lower specification limit, respectively. When `USL` $= 3\sigma$ for a process, the specification limit is referred to as a control limit (`UCL` and `LCL`, respectively). Similarly, when `USL` $= 2\sigma$ for a process, the specification limit is referred to as a warning limit (`UWL` and `LWL`, respectively). For our experiments, the use of either warning or control limits is too relaxed for the reactive granularity of multimedia systems. Instead, we have found that stream-dependent specification limits work well. For example, to support lip-synchronous video, a tolerance specification of $33ms$ may be acceptable. On the other hand, to support audio-annotated slide shows, a tolerance specification of $500ms$ may be acceptable. Each tolerance specification associates different statistical likelyhood to the presence of asynchrony outliers (those that exceed the tolerance region — assuming $\epsilon$ is

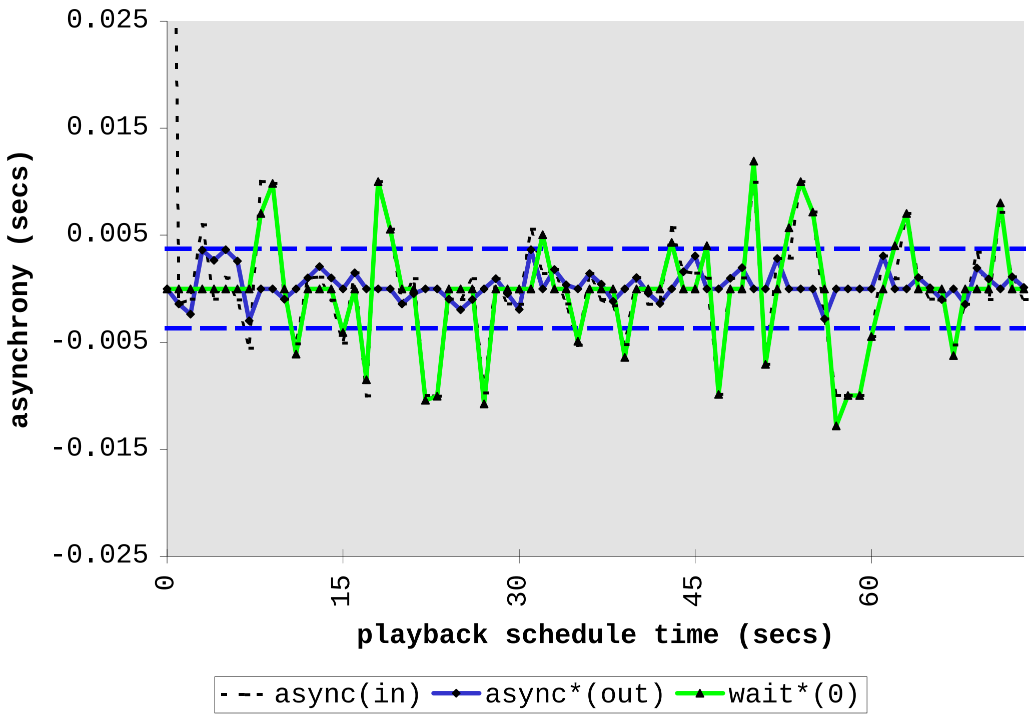


Figure 8.12: Tightened synchronization measures are specifiable with policy P5 by reducing the tolerance region of a temporal relationship.

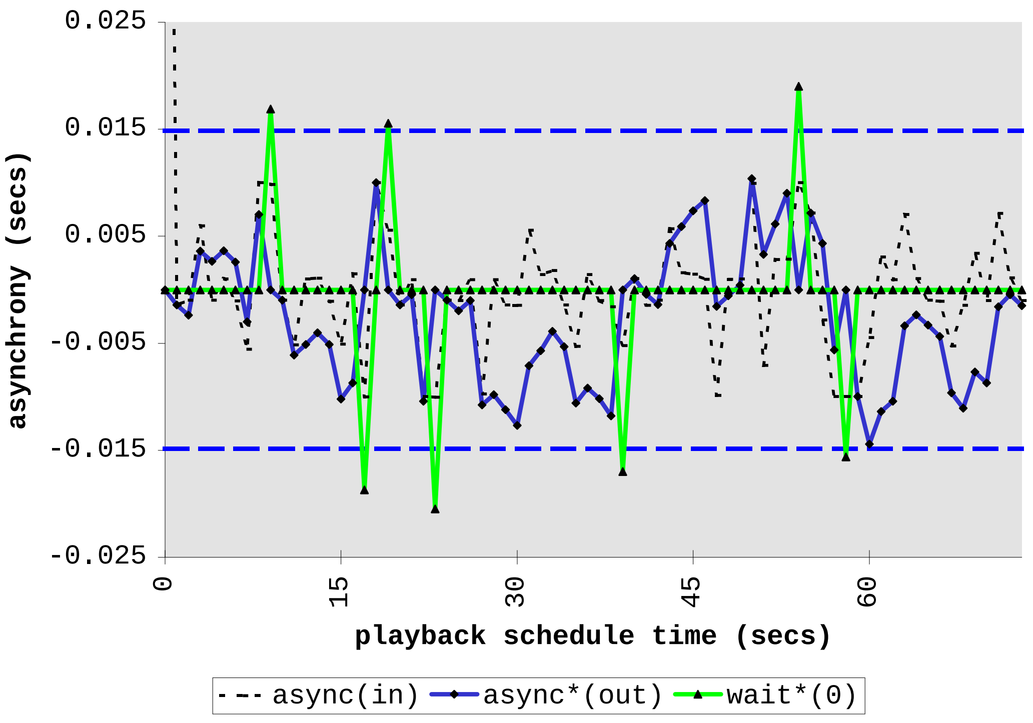


Figure 8.13: Relaxation of synchronization by policy P5 by expanding the tolerance region of a temporal relationship. The net effect of P5 is that (1) *asynchrony floats* between LSL and USL) control limits while (2) playback discontinuities due to synchronization occur only when the asynchrony exceeds such limits.

a normal variable). Asynchrony outliers trigger synchronization measures and thus a loss on playback continuity.

Synchronization and continuity tradeoffs are specifiable. For example, Fig. 8.12 shows how these mechanisms are used to enforce conservative handling of the asynchrony. This is accomplished by reducing the tolerance region of the temporal relationship between $A$ and $D$. On the other hand, Fig. 8.13 shows how these mechanisms are used to enforce relax handling of the asynchrony. This is accomplished by expanding the tolerance region of a temporal relationship between $A$ and $D$.

The net effect of P5 is as follows:

- $LSL \leq \mu_{async} \leq USL$
  *asynchrony floats* between `LSL` and `USL` control limits while
- $(wait_{async} \leq LSL) \,||\, (USL \leq wait_{async})$
  playback discontinuities due to synchronization occur only when the asynchrony exceeds such limits.

Consequently, tradeoffs between asynchrony and continuity are now specifiable: (1) the asynchrony $\epsilon_i$ is controlled by the tolerance region ($|USL - LSL|$) and (2) discontinuities are now characterized by the probability of the asynchrony exceeding such tolerance $2P(|\epsilon| > USL)$. The P5 mechanism should be coupled with adaptive scheduling (as in P3) to remove the presence of long-term playback asynchrony trends.

## 8.7 Concluding Remarks

The playback of re-executable content streams, together with continuous streams, arises during the streaming of inputs to multiple applets in a workspace. This problem is also of strong interest to next generation multimedia frameworks as found for ubiquitous computing, nomadic computing, wireless ATMs, and ATM IPng technologies with the introduction of time-dependent streaming of re-executable content to remote appliances. The media integration and streaming mechanisms on these frameworks face media heterogeneity. Streaming and integration require us to characterize stream requirements. Whereas for continuous media streams we possess a strong characterization of their requirements, on the other hand, for re-executable content streams, we lacked a suitable characterization that addressed both synchronization and continuity. This is a primarely due to the burstiness, statefulness, and unpredictability of these new and (yet to be defined) re-executable content streams.

In this chapter, I described mechanisms to characterize the playback of such heterogeneous streams. These mechanisms address both synchronization and continuity as part of its design, not as an afterthough to a continuous media streamer. The mechanisms are based on periodical monitoring (loose supervisory controls) coupled with statistical process controls to induce a long-term effect over adjustments made to any of the streams. When the *asynchrony* $\epsilon_i^*$ follows a long-term trend, SPC-based monitoring is used to detect the trend and adaptive scheduling discussed in Chapter 4 is used to remove the trend. The mechanism is somewhat more forgiving of short-term timing departures. Through the characterization of long-term synchronization relationships between the streams in a workspace, the use of long-term policies such as P3 as shown in [66] or P5 can reduce asynchrony and still preserve continuity.

# CHAPTER 9

# ON DEVELOPING REPLAYABLE APPLETS

*"Please pay no attention to the man behind the curtains..."*
— *from The Wizard of Oz by Frank Baum.*

## 9.1 Introduction

The integration of an applet into a replayable workspace is referred to as replay-awareness. Two middleware layers: the media layer and the workspace layer (see Fig. 9.1) provide APIs for compliance to replay-awareness. The underlying middleware (i.e., tool coordination and media integration) mechanisms was analyzed in Chapters 6 and 7, respectively. In this chapter, I focus instead on key issues on the design and implementation of replayable applets.

The workspace layer provides compliance to tool coordination mechanisms. The workspace API consists of applet callbacks (such as record() and replay()). The various workspace API callbacks are to be implemented by the developer of a replayable applet developer.

The media layer provides compliance to media integration mechanisms. The media API consists of media-independent primitives that operate over the shared data model presented in Chapter 6. The developer of a replayable applet must define the contents of events in an applet stream.

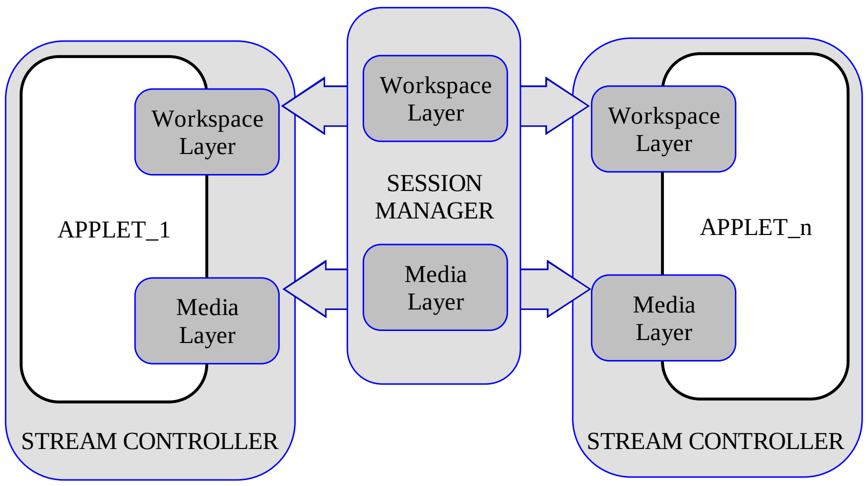


**Figure 9.1: The compliance layers of a replayable workspace. An applet interfaces to the replayable workspace through the session manager. Compliance to replay-awareness is achieved through two layers: (1) media layer and (2) workspace layer. Media layer compliance allows the session manager to synchronize the playback of multiple applets. Workspace layer compliance allows the session manager to orchestrate the applets.**

## 9.2 Criteria for Replayable Applets

Criteria to understand which applets can be made replayable is of interest. Here, I characterize the sort of applets that can be extended into temporal-awareness.

Temporal-awareness is extended to an applet by means of a temporal controller. The temporal controller assumes the computation component of an applet to have a well-defined abstract base machine (*abm*). The following are characteristics of particular *abm*s that facilitate capture and replay:

- sampling points:

  that is, the applet should have a well-defined abstract base machine with a single point of control (e.g., a central processing loop). This facilitates sampling and dispatching of *abm* operations. On the other hand, for example, an applet with multiple sampling points could easily lead to either undersampling or oversampling of applet inputs.

- self-contained state:

  that is, the state $S_j$ of the abstract base machine can be completely checkpointed (i.e., snapshot and stored). Similarly, it is possible to install the state snapshot at a later time.

- determinism:
  that is, applying the same event $e_i$ over the same state $S_j$ on the same abstract base machine produces the same next state $S_{j+1}$.

The closer an applet's *abm* resembles an extended finite state machine model:

$$S_{i+1}(abm) = S_i(abm) + abm(e_i) \tag{9.1}$$

then the greater likelyhood of successfully introducing temporal awareness into the applet.

## 9.3 The Temporal Controller

In this section, I specify the temporal controller and its interfaces for extending an applet into temporal-awareness.

### 9.3.1 Interfaces to an Applet

A temporal controller associates a temporal media stream (aka applet stream) to an applet. The temporal controller shields its applet from awareness about this temporal media stream, as follows.

Two simple primitives (fetch() and write()) provide the coupling between a temporal controller and its applet stream (see Fig. 9.2). These primitives operate over the shared, media-independent, model for applet streams presented in Chapter 6.

- fetch( $E_i$ ):
  This primitive is called by the temporal controller of an applet to retrieve the next event sequence from an applet stream.
- write( $E_i$ ):
  This primitive is called by the temporal controller of an applet to write the next event sequence as a persistent object into its associated applet stream.

Two simple primitives (record() and replay()) provide the coupling between a temporal controller and its underlying applet (see Fig. 9.2). These primitives operate over the shared, media-independent, model for applet streams.

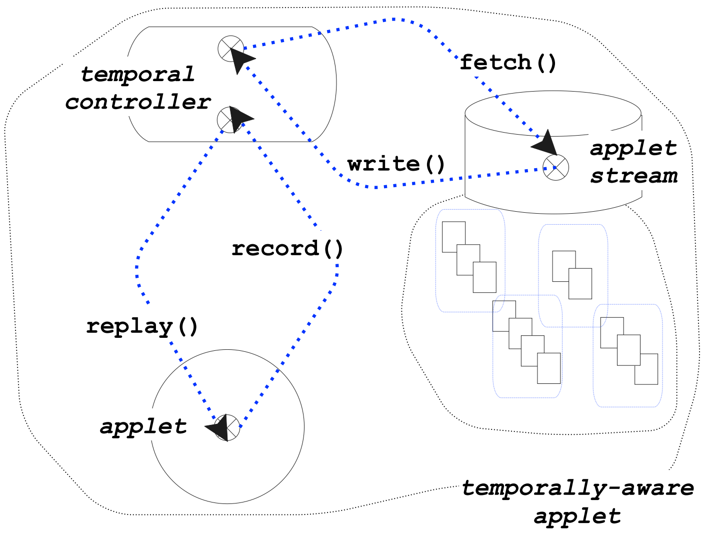


**Figure 9.2: The temporally-aware applet and its interfaces.**

- record( $e_i$ ):

  This primitive is called by the temporal controller of an applet to sample the next event from the abstract base machine of its underlying applet.

- write( $e_i$ ):

  This primitive is called by the temporal controller of an applet to present the next event into the abstract base machine of its underlying applet.

### Discussion

The reader may raise two issues of interest here. First, operations over a media independent model could lead to poor performance on the fetch() and write(). Here the answer is interesting, media access operations need to be scheduled too (i.e., prefetching and buffered writes). Although research on batched access for real-time playback of continuous media is found in [87, 81], the batching of heterogeneous streams, it is usually discarded as non-bandwidth critical. However, some re-executable content streams are bandwidth critical (for example, consider an incoming display stream from a remote instrument). Clearly, media access needs to be configurable to stream requirements 9.3. In the next section, I discuss block prefetching and writing to achieve efficient media access over the audio stream under my media-independent data model.

The second issue is more subtle and arises from experience with our paradigm as

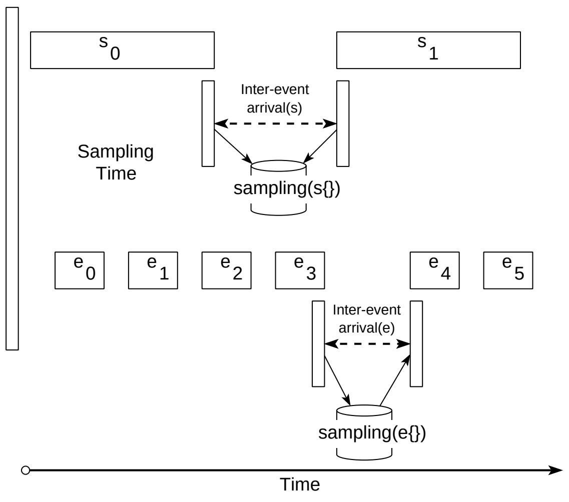


**Figure 9.3: Media access needs to be configurable and efficient. For example, either large sampling overheads; asymmetrical sampling and fetch overheads; as well as asymmetrical stream sampling overheads could introduce playback faithfulness departures.**

reported by Dr. T. Weymouth, a principal investigator in the Medical Collab. How does non-sequential access (for example, as induced from browsing of a session) affect these fetch() and write()? The answer lies in that the representation of a session should not be a linear structure but rather a tree structure (i.e., as induced by branch events). To support efficient access over this data structure, media access operations need to operate over a coarse event granularity. In the next section, I discuss how media access operations rely on event sequences provide the grain for amortizing media access overheads.

## 9.4 Guidelines

The following are general guidelines for designers of replayable workspaces, designers of temporal controllers as well as for developers of replayable applets.

### 9.4.1 Guidelines for Designers of Replayable Workspaces

1. Determine the tool services that the workspace ought to provide. Tool services are usually domain-dependent (e.g., computer-supported radiology, space science, software development). Some tool services such as audio/video annotation as well as gesturing are common across domains.

2. Determine the mapping of services to applets. Ideally, each applet should provide one service to the workspace. This way, its abstract base machine is simplified.

3. Specify the relationships between applets on a workspace. For every applet pair in a workspace determine whether a relationship exists and specify the minimum tolerances for each of these relationships.

### 9.4.2 Guidelines for Designers of Temporal Controllers

1. Implement the translation of workspace API commands. Since the session manager relies on a polymorphic API, each applet must translate these API commands into the respective applet-based operations. The temporal controller of an applet encapsulates these translations. For example, consider an audio applet, the workspace replay() API command could be translated into the applet-based audio.play() operation.

### 9.4.3 Guidelines for Developers of Replayable Applets

1. Define the abstract base machine of the applet. The definition of the applet's *abm* also implies the definition of events, event sequences, and the characteristics of the applet stream associated to the applet's *abm*.

2. Determine the interfaces to the selected *abm*. Ideally, one sampling point should provide full sampling of *abm* inputs and state. Similarly, one insertion point should allow insertion of events and/or state into the applet's *abm*.

3. Customize media access parameters to support efficient I/O of event sequences in an applet stream. For example, on the NeXT platform, I found that for the audio stream, prefetching four frames at a time preserved playback continuity while maintaining low fetch overheads. On the other hand, I found that the large segments of the window stream could be prefetched at a single time with amortizable overhead.

## 9.5 Replayable Window and Audio Streams

In this section, I discuss issues on retrofitting a monolithic application into replay-awareness. The application, a MacDraw-like object-oriented drawing application was chosen. The application was capable of tasks such as drawing figures, entering text, moving figures, grouping objects, resizing objects, scribbling, font and color selection, etc..

### 9.5.1 Defining Tool Services

In this prototype, we desired to extend replay-awareness services to allow the drawing application to record and replay an audio-annotated session with the drawing application.

### 9.5.2 Mapping Services to Abstract Base Machines

A drawing application is composed of one or more canvases and a toolbox of drawing elements (e.g., circles, squares, freeform, etc.). A set of options panels (accessible through menus) associate options to the "*look-and-feel*" of each drawing element. Such drawing application is characterized by a strong coupling of user interface to computation. In essence, the drawing application computation are its canvases. One feasible abstract base machine for the representation of such a workspace is the set of user interactions over canvases on the workspace. The playback user interactions can be achieved through the playback of the window stream. The stand ard abstract base machine associated with the window stream represents a good choice here.

The later-time collaboration content of a session can be augmented through audio-annotation. Audio has an inherent high degree of temporal awareness. The standard abstract base machine associated with continuous audio represents a good choice here.

### 9.5.3 Defining Temporal Relationships

The playback of a session is to be annotated with continuous audio. The playback of such a session requires to: (1) record and re-execute window events (e.g., gesturing, typing, moving windows); (2) record and replay audio-annotations; (3) synchronize the playback of these temporal streams; and (4) support the playback of one or both streams.

In terms of synchronization relationships, audio and window stream hold only a temporal (i.e., non-causal) relation. Since we desire to meet the stringent tolerances of audio playback, it is desirable to adjust the playback of the window stream (W) to the audio stream (A). Equivalently, we desire to preserve the one-way, temporal relationship: $R(A \leftarrow W)$.

### 9.5.4 Defining the Abstract Base Machines

The window stream is associated to the abstract base machine of: MouseDown(), MouseUp(), MouseDragged(), MouseMoved(), WindowMoved(), WindowResized(), MouseEntered(), MouseExited(), CursorUpdate(), TimerEvent(), etc.. The audio stream is associated to the abstract base machine of continuous audio processing: record(), replay(), pause(), continue(), save(), open(), stop(), etc..

### 9.5.5 Events and Event Sequences

The window stream is composed of X-events that represent abstract base machine operations. These events are arranged into event sequences that capture the semantics of mouse clicks: MouseClick() = MouseDown(), MouseDragged()*, — MouseMoved()*, MouseUp(), The audio stream is composed of audio frames. These audio frames are accessed by abstract base machine operations.

### 9.5.6 Architecture

This prototype supported two streams: the window stream and the audio stream. Each stream was dispatched to a separate processor. During playback, the window event stream was dispatched to the CPU for re-execution whereas the audio stream was playback by the DSP — assumed to be dedicated. These components (application, streams, DSP, CPU, infrastructure services, disk, and data paths) are shown in Fig. 9.4.

### 9.5.7 Compliance to Temporal-Awareness

#### Fetch

The fetch of an event is performed through the getObjectAt[] method. This call accesses the specified indexed location *physicalIndex* on the specified stream *streamSelector* and returns the header to the associated event *trackElemHeader*.

```
- (TRACK_ELEMENT *) getObjectAt: (unsigned int)physicalIndex
                      onStream: (STREAM_SEL)streamSelector {
    [accessLock lock];
        getTrackUsedFor(selectedTrack,streamSelector);
        trackElemHeader = (TRACK_ELEMENT *)
```

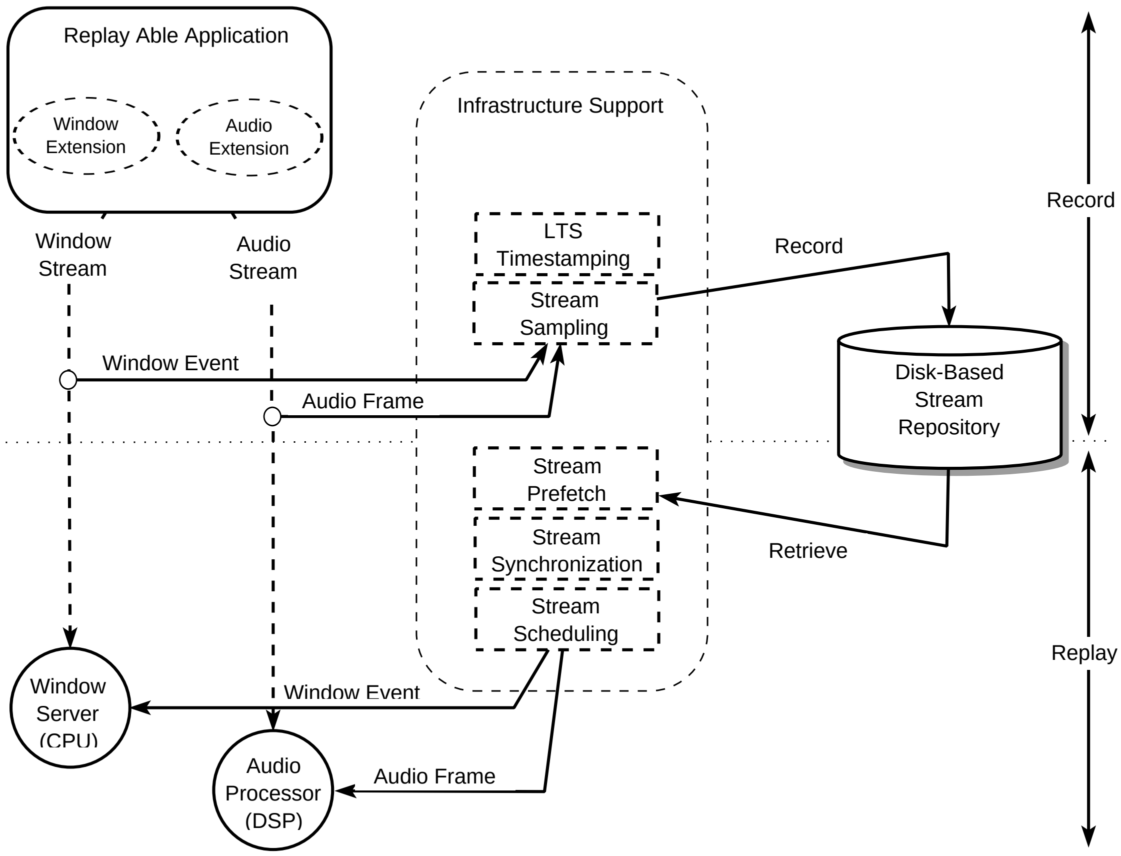


**Figure 9.4: Complete view of the architecture of the monolithic replayable drawing applet showing both record and replay infrastructure services.**

```
                          [selectedTrack elementAt: physicalIndex];
    [accessLock unlock];
}
```

Streams with BLOB (large binary objects) events can rely on our data model for deferential access. In the case of the audio stream, the getObjectAt[] method returns a BLOB file reference (*audioFile*. The BLOB reference is stored as a relative path into the session object directory. An stream handler readAudio[] explicitly allocates the required memory (*AudioBuffer*) into which it expands the de-referenced BLOB object.

```
- readAudio: (char *) audioFile : (SNDAudioStruct **) AudioBuffer {
    SNDReadAudiofile(audioFile, toAudioBuffer);
}
```

## Write

The write of an event is performed through the recordInto[] method. This call inserts the specified event *object* into the specified stream *streamSelector*. The method associates a timestamp *aTime* and a persistent object *trackElemHeader* to the event.

```
- recordInto: recording thisObject: object
                          onStream: (STREAM_SEL)streamSelector
                            atTime: (TIME_INDEX)aTime {
    trackElemHeader.object = object;
    trackElemHeader.time = (TIME_INDEX) aTime;
    getTrackUsedFor(selectedTrack,streamSelector);
    [accessLock lock];
        if(selectedTrack) [selectedTrack addElement: (void *)&trackElemHeader];
    [accessLock unlock];
}
```

Streams with BLOB events can rely on our data model for deferential access. In the case of the audio stream, the recordInto[] method associates a BLOB file reference *file[i].name* to a BLOB object. The BLOB reference is stored as a relative path into the session object directory. An stream handler writeAudio[] explicitly stores the audio frame (*sndBuffers[i]*) into a BLOB object.

```
- writeAudio: (int) tag {
    SNDWriteAudiofile( (char *) files[tag].name,
                       (SNDAudioStruct *) sndBuffers[tag] );
}
```

### Record

Record relies on sampling functions to intercept *abm* operations. The guiSamplingFunction[] samples an X-event *anEvent*, timestamps it *ltsNOW*, and requests the event to be written to disk recordOnThis[].

```
guiSamplingFunction( NXEvent *anEvent ) {
    eventCounter++;
    if(LooseEvent(anEvent))
        return nil;
    getTimeStamp(&ltsNOW);
    newObj = [myHistClient recordOnThis: myRec withLTS:ltsNOW ...];
    [mySelf ackRecordedEntry];
}
```

The recordNewAudioFrame[] records a new BLOB object *aNewAudioFrame* (i.e., the next audio frame). Since audio stream own sampling function SNDStartRecording[] is used to sample the audio frames.

```
- recordNewAudioFrame: sender {
    audioCount = audioCount + 1;
    aAudioTag = aAudioTag + 1;

    selectedBuffer = (unsigned int) [self removeFromAvailBuffers];
    aNewAudioFrame =  sndBuffers[selectedBuffer];
    tasks[selectedBuffer].taskCode = (int) SAVE_AUDIO_SAMPLE;
    tasks[selectedBuffer].audioTag = (int) aAudioTag;
    sprintf(files[selectedBuffer].name, "%s%d", [self FSrecAudios], audioCount);

    SNDStartRecording( aNewAudioFrame, selectedBuffer,
                       SOUND_UPDATE_PRIORITY,
                       NO /* PRE-EMPTION */,
```

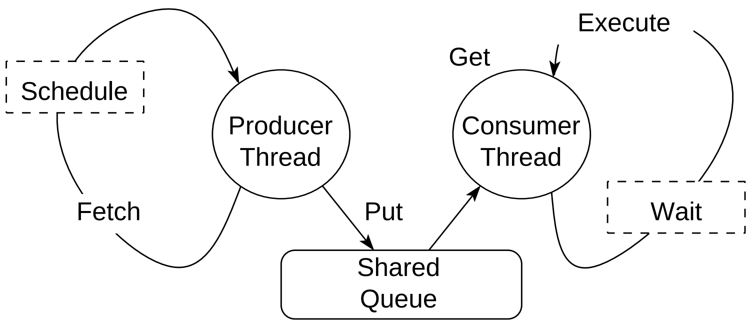


**Figure 9.5: Thread model for the replay of the window-event stream. The producer thread fetches and dispatches events into the event queue. The consumer thread manages asynchronous re-execution of window events that were posted into the shared event queue.**

```
                    (SNDNotificationFun) SND_NULL_FUN,
                    (SNDNotificationFun) recEndFunction );

    [self addToBuffersInUse: selectedBuffer];
    recordCount++;
}
```

### Replay

Streams playback in separate threads. Figure 9.5 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 time-stamps of events 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.

### Optimizing Sampling and Prefetching

Figure 9.6 shows the thread model used to replay audio frames. To reduce overheads in disk I/O access of audio frames, buffered sampling and prefetching of multiple frames is used (i.e., block-based writing and prefetching, as also found in CMSS [89]).

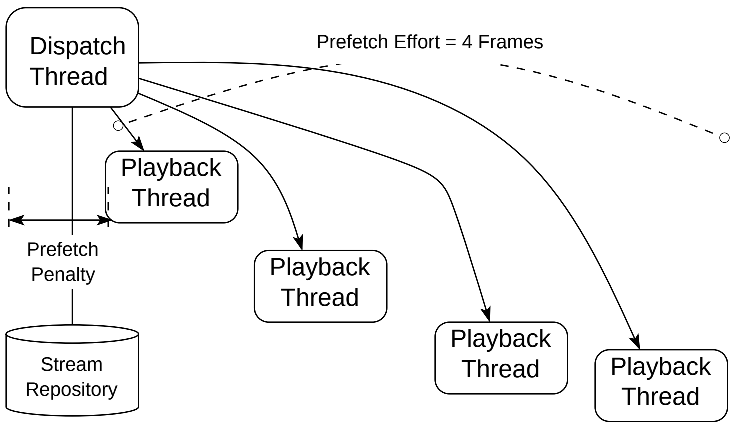


**Figure 9.6: Thread model for replay of audio-frames. The dispatch thread amortizes prefetch overhead among several audio frames. Audio frames execute asynchronously on the audio device.**

I baselined three important parameters that affect overheads $f_o(A1)$ during the record and replay of audio frames at the application layer: (1) the audio frame size, (2) the buffering effort for writes, and (3) the buffering effort for prefetches.

The baselined values used to optimize the handling of the audio stream are summarized next.

- 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 4.2). 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.

Each read request for an audio frame readAudioFrame[] is handled by the prefetch manager, which loads the frame into memory from either the file system readAudio[] or if successful, from a prefetched buffer *aP = (PREFETCH-ENTRY *)*.

```
- (SNDAudioStruct *) readAudioFrame: (unsigned int) tagIndex {
    [PlayDispatchMutex lock];
```

```
        aNewAudioFrame = [self prefetchAudioFrame: (unsigned int) tagIndex];
    [PlayDispatchMutex unlock];
}

- (SNDAudioStruct *) prefetchAudioFrame: (unsigned int) tagIndex {
    tagIndex--;
    aP = (PREFETCH_ENTRY *) [self tagToBufferMapping: tagIndex];

    if ( aP ) {
        selectedBuffer = aP->inWhichBuffer;
    } else {
        nextObject = [myAudioTrack objectAt: tagIndex];
        if( nextObject ) {
            audioFileName = [nextObject getAnnotation];
            selectedBuffer = [self removeFromReplayAvailBuffers];
            [self addToReplayBuffersInUse: selectedBuffer :tagIndex];
            [self readAudio: audioFileName : &sndBuffers[selectedBuffer]];
        }
        else
            return (SNDAudioStruct *) 0;
    }
    return sndBuffers[selectedBuffer];
}
```

Each write request for an audio frame readAudioFrame[] is handled by the buffered write manager, which buffers frames in memory *aP* until it gets a sufficiently large block of frames ( *WRITE-EFFORT* to write them to the file system (taskManager[] is a write call).

```
- periodicalWritingCheck: (int) tag {
    if( (tag % HOW_OFTEN_TO_WRITE ) == 0) {
        [recordInUseLock lock];
            i = 0;
            writeCount = 0;
            while( writeCount < WRITE_EFFORT ) {
                ...
                aP = [recordAudioInUse elementAt: i];
                if (aP) {
```

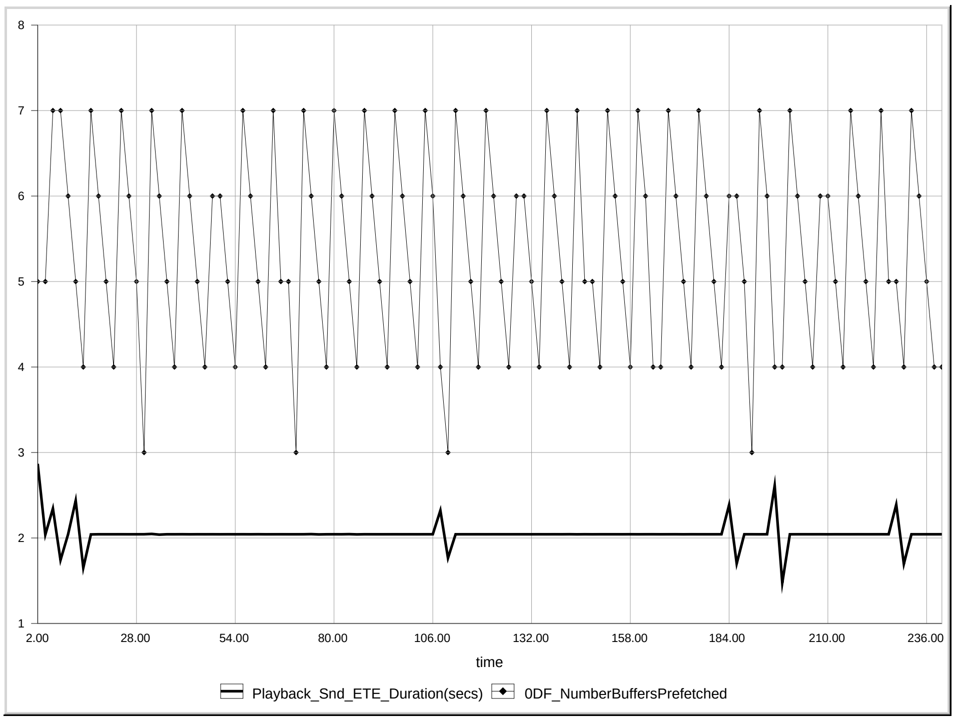


**Figure 9.7: Behaviour of block prefetching over the continuous playback of the audio stream.**

```
                if( aP->stillInUse == YES ) {
                    selectedBuffer = aP->inWhichBuffer;
                    returnCode = [self taskManager: selectedBuffer];
                    if( returnCode ) {
                        [recordInUseLock unlock];
                            [self removeFromBuffersInUse: i];
                            [self addToAvailBuffers: &selectedBuffer];
                            writeCount++;
                        [recordInUseLock lock];
                    }
                }
            } else break;
            i++;
        }
    [recordInUseLock unlock];
  }
}
```

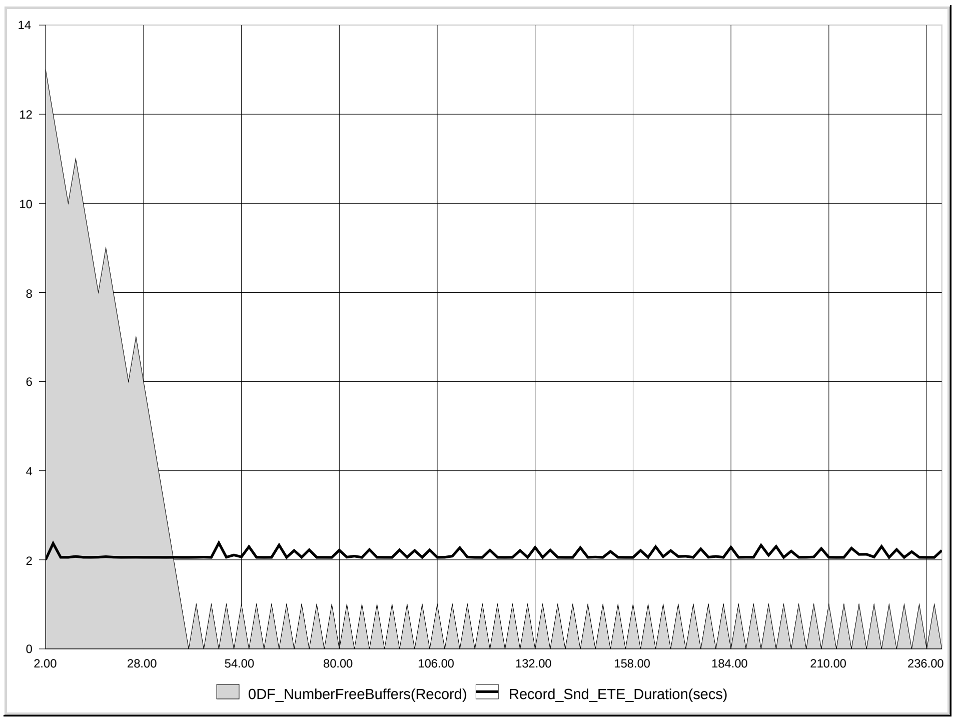


**Figure 9.8: Effect of block buffered writes over the continuous sampling of the audio stream.**

Currently, both prefetching and block writing components of the audio temporal controller are media-dependent. Their performance *wrt* the continuity of audio playback is shown in Figs. 9.7 and 9.8. However, both components could easily be developed into media-independent components for any continuous media stream. Adjusting such components to meet the tolerances of different continuous media streams would correspond to fine tuning the block-effort to amortize fetch and write overheads.

## 9.6 Integration of Audio, Window, and Video Streams

In this section, I consider a different angle of the retrofitting problem. This time, I analyze the formulation of the media integration requirements for a workspace composed of multiple streams: audio (A), video (V), and window (W) streams, where there are both causal and temporal relationships between these streams.

### 9.6.1 Requirements

The goal is to support faithful playback of voice-annotated video and user-interactions. This goal requires addressing the following issues.

- First, user interaction requests over the video stream are asynchronous and are subject to timing variability. For example, consider a session on which a radiologist zooms-in during a specific MRI video segment and proceeds to gesture and voice annotates certain features of interest. Because the re-execution of session objects is subject to unpredictable timing variability, departures in the schedule observed during the original capture of the session will be observed.

- Second, user interaction requests introduce data dependencies over the video stream (i.e., a user interaction request acts over *specific* data on the video stream). If the data required to satisfy a request is not present, the correctness of the re-execution may be lost.

As stated before, the media integration solution has two mechanisms: (1) *inter-stream synchronization* and (2) *adaptive scheduling*. The inter-stream synchronization mechanism will be used to preserve relative timing *wrt* the playback of the audio stream and to preserve data dependencies between the window and video streams. The adaptive scheduling mechanism will be used to compensate per-stream asynchrony trends so as to provide (best-effort) smooth and continuous playback. This is accomplished through the use of statistical-based run-time modifications to a resource on a slave stream used as a degree of freedom resource. In this section, I show how to use these mechanisms to address our requirements.

### 9.6.2 Modeling

A scheduling interval on the video stream consists of a discrete sequence of $n$ video frames $(v_1, \cdots, v_n)$ found between two consecutive synchronization events $s_i$ and $s_{i+1}$. A fundamental parameter in describing the playback quality of the video stream is the presentation *frame rate* or just $v_f$. The frame rate $v_f$ is measured in *frames per second* (*fps*) and it represents the number of frames to a scheduling interval.

The parameter $v_f$ affects the playback quality of the video stream QoS(V). In general, during replay, increasing the $v_f$ value increases QoS(V). Similarly, decreasing the $v_f$ value decreases QoS(V). However, increasing the $v_f$ value over a certain value $v_{max}$ results in no further increases in QoS(V). The $v_{max}$ playback value is inherently preset by the actual frame rate used to record of the video stream. We will use $v_f$ as the degree of freedom resource for handling a video stream.

### 9.6.3 Inter-stream Synchronization Mechanism

In Chapter 4, I discussed media integration support needed to preserve the temporal correspondence between window and audio streams, $R(A \leftarrow W)$. In Chapter 7, I generalized media integration support for multiple heterogeneous streams and their relationships.

As before, to preserve the relative timing of audio and video streams, synchronization relationships are used. Here, lets use the synchronization relationship $R(A \leftarrow VW)$; decomposable as:

- $R_1(A \leftarrow W)$
  that is, the playback of voice-annotated user interactions.
- $R_2(A \leftarrow V)$
  that is, the playback of voice-annotated video.

During the capture of the session, periodically, a synchronization event $s_i(a_j \leftarrow v_k)$ is posted by the audio stream for each relationship. The synchronization event is introduced into each *slave* stream of $A$ (e.g., video and window streams). During playback of the session, the scheduling of the synchronization event $s_i(a_j \leftarrow v_k)$ by the schedulers for the video and window streams triggers a dependency constraint *wrt* the audio stream.

Similarly, *pair-wise* inter-stream asynchrony components for each synchronization relationships are computed (see Fig. 9.9) as follows:

$$\epsilon_i^* = \left\{ \begin{array}{lll} \epsilon(A,W) & = t(s_i(A)) - t(s_i(W)) & for(A \leftarrow W) \\ \epsilon(A,V) & = t(s_i(A)) - t(s_i(V)) & for(A \leftarrow V) \end{array} \right\} \tag{9.2}$$

As before, smoothed asynchrony estimates are used to guide the statistical process controls of each of the slave stream schedulers.

Figure 9.9 illustrates the use of partial synchronization events $(A \leftarrow W)$ and $(A \leftarrow V)$ so as to accomplish the integrated replay of window, audio, and video streams. It also shows that because stream schedulers are independent, each slave stream (i.e., $V$ and $W$) follows independent scheduling intervals *wrt* each other. Figure 9.9 also illustrates:

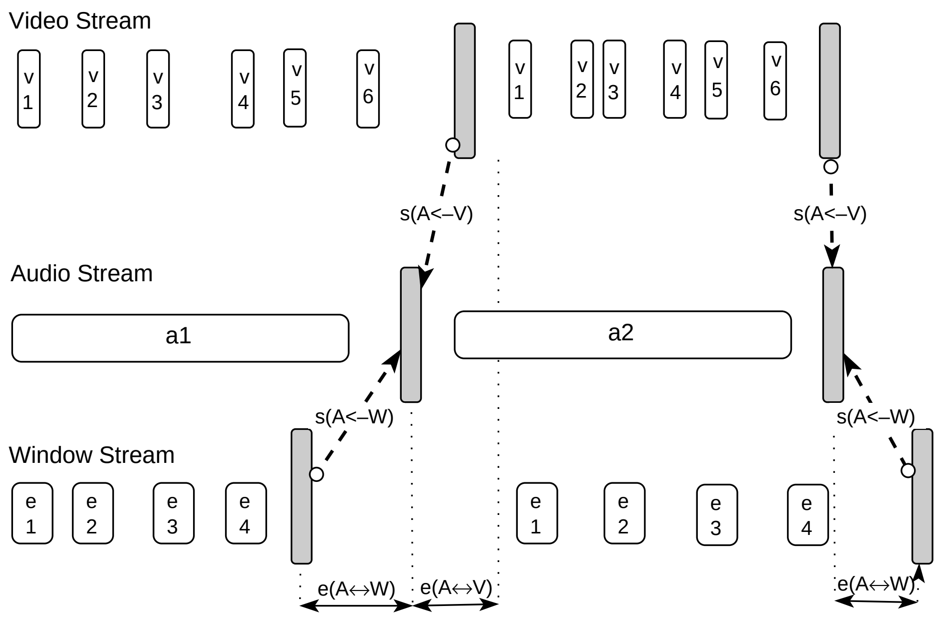


**Figure 9.9: Inter-stream synchronization for integrated digital audio, video and window streams. A synchronization event, $(A \leftarrow VW)$, preserves relative synchronization between our $A$ (audio), $V$ (video), and $W$ (window) streams. The inter-stream asynchrony between corresponding synchronization events is variable. It is estimated by the substraction of observed schedule times for synchronization events at each stream — e.g., $\epsilon_i = \{t(s_i(A)) - t(s_i(W)), t(s_i(A)) - t(s_i(V))\}$.**

| Protocol | Video Ahead Audio Treatment | Video Behind Audio Treatment |
|---|---|---|
| P3: 1 Way Adaptive | (1) wait for matching audio event (freeze last frame) (2) if this is a trend (to be ahead), compensate by decreasing video frame rate. | (1) if this is a trend (to fall behind), compensate by increasing video frame rate. |

**Table 9.1: Comparison of synchronization protocols. The first column lists the protocol name, followed by the specified handling of inter-stream asynchrony. The video stream scheduler implements the treatment.**

- the use of synchronization events to measure the inter-stream asynchrony between audio, window, video. The inter-stream asynchrony between window and audio streams is measured by $e_i(A \leftarrow W)$. The inter-stream asynchrony between video and audio streams is measured by $e_i(A \leftarrow V)$.
- the independence of the two pair-wise components $(AW)$, $(AV)$. Both $e_i(A \leftarrow W)$ and $e_i(A \leftarrow V)$ pair-wise components may have different magnitudes and directions.

**Discussion**

In conclusion, the audio enjoys uninterrupted playback (for example, because it does not initiate synchronization) whereas the video stream (the synchronization requester/initiator) is manipulated using a release mechanism (for example, its previous frame can be repeated when running ahead) as summarized in Table 9.1.

### 9.6.4 Adaptive Scheduling Mechanism

The adaptive scheduling mechanism presented in Chapter 4 is based on the compression and expansion of a stream's resource. The resource is used as a degree of freedom to compensate for trends in inter-stream asynchrony. For the window stream, the selected resource was the inter-event delay times, for the audio stream a good choice is the duration of silence segments (as in [31, 74]), and for the video stream, our choice is the playback frame rate, as suggested in [76].

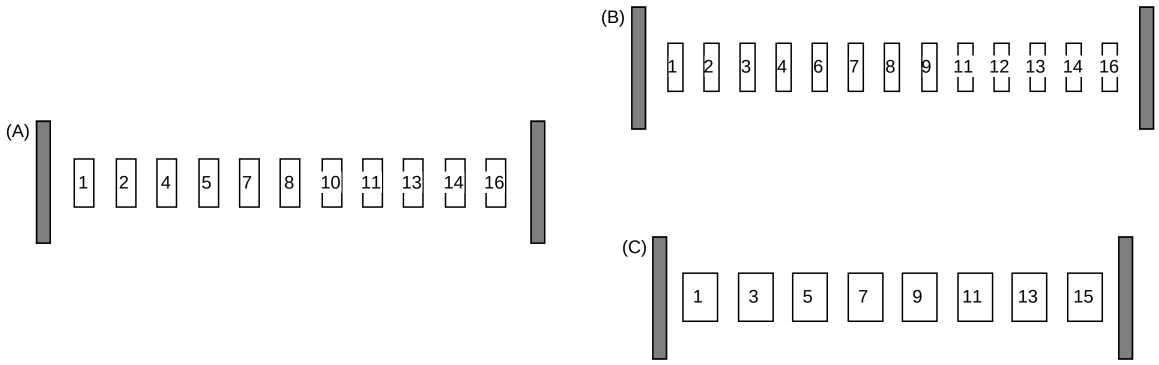


Figure 9.10: Adaptive video scheduling. Time line diagram shows potential adjustments to a scheduling interval of the video stream. In this figure, a maximum video frame rate $v_{max}$ of $16fps$ is assumed. Part (a) shows the current frame rate $v_f = \frac{2}{3}v_{max}$. One-third of all video frames in this interval are dropped. Part (b) shows a potential adjustment to the frame rate $v_f = \frac{4}{5}v_{max}$ for when the video stream scheduler detects a trend to lower overheads during playback. Correspondingly, an attempt to ***smoothly increase*** the QoS of the video stream is made. Part (c) shows a potential adjustment to the frame rate $v_f = \frac{1}{2}v_{max}$ for when the video stream scheduler detects a trend to higher overheads during playback. Correspondingly, an attempt to ***smoothly decrease*** the QoS of the video stream is made.

Figure 9.10 shows that adaptive scheduling compensations to the frame rate resource for the video stream. In this figure, a maximum video frame rate $v_{max}$ of $16fps$ is assumed. Figure 9.10(a) shows the *current* frame rate $v_f = \frac{2}{3}v_{max}$. One-third of all video frames in this interval are dropped. Figure 9.10(b) shows a potential adjustment to the frame rate $v_f = \frac{4}{5}v_{max}$ for when the video stream scheduler detects a trend to lower overheads during playback. Correspondingly, an attempt to *smoothly increase* the QoS of the video stream is made. Figure 9.10(c) shows a potential adjustment to the frame rate $v_f = \frac{1}{2}v_{max}$ for when the video stream scheduler detects a trend to higher overheads during playback. Correspondingly, an attempt to *smoothly decrease* the QoS of the video stream is made.

#### Discussion

Here again, the reliance on statistical process controls for the implementation of the scheduling mechanism extends several existing solutions for long-term control of the asynchrony and smoothing of the compensation process. First, my approach provides best-effort adjustments to QoS(V) that are smooth and conservative; based on sound statistical principles. Second, as with audio and window streams, these conservative adjustments lead to a process control over the asynchrony which has statistical properties. Last, these long-term mechanisms enable efficient application-layer control of the inter-stream asynchrony so as to allow integrated replay of heterogeneous streams such as window, audio, and video streams over a range of load conditions.

### 9.6.5 Preserving Data Dependencies

Previously, I discussed the synchronization support between window and audio streams. However, note that audio and window streams exhibit no data dependencies that affect the correctness of their re-execution. The support of the replay of synchronized user interactions over the playback of video introduces data dependencies between the window and video stream. This is a causal synchronization relationship specified as $R(V \leftarrow W)$.

Let's define two new streams: $R^*$ and $R$. The stream $R^*$ consists of $n$ events $(r_1^* \cdots r_n^*)$. These events represent data references to video frames observed during the original capture of a session. That is, each $r_i^*$ *refers* to some $v_j^*$. The stream $R$ also consists of $n$ events $(r_1 \cdots r_n)$. These events represent the data references to video frames intercepted during replay. That is, each $r_i$ *will refer* to some $v_j$.

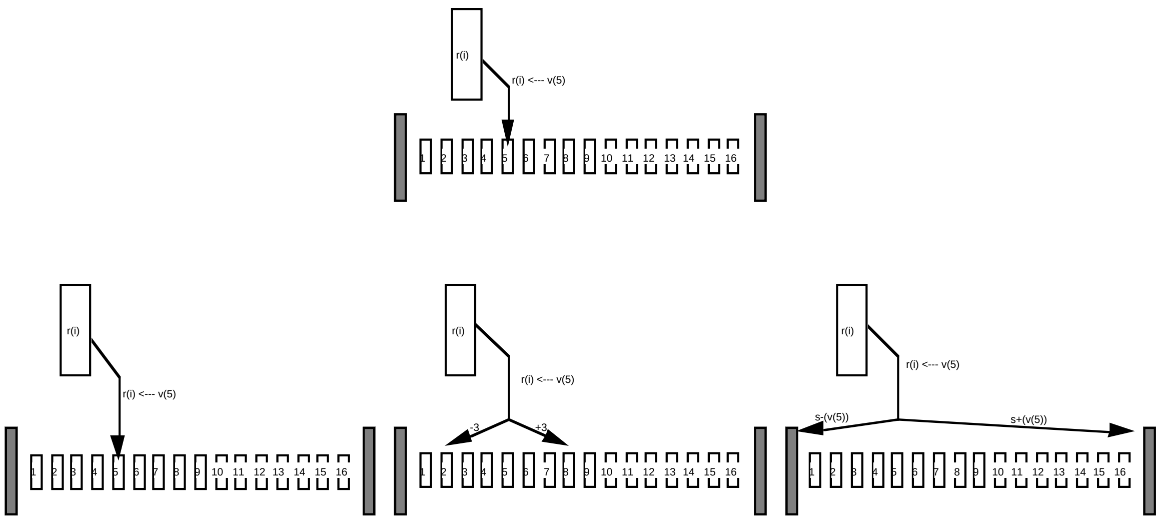


**Figure 9.11: Not all interaction requests with the video stream *require* synchronization to be performed *wrt* the exact same frame as in the original session. The left side shows the representation of a recorded interaction request $r_i$ over the video stream at frame $v_5$. The right side shows possible enforcements for this request during the re-execution of the interaction, based on the target frame precision of the original request.**

If we are to preserve data dependencies, our goal now becomes to enforce a fine-grained synchronization relationship that allows each $r_i$ to refer to a *suitable* video frame $v_j$ (i.e., a suitable replacement for video frame $v_j^*$).

### 9.6.6 Adaptive Protocols for Data Dependencies over Video

During replay, we can take advantage of some characteristics of the video media to facilitate the enforcement of data dependencies. In particular, the video stream is highly redundant and stateless. Next, we discuss how these characteristics facilitate our tasks.

Not all interaction requests with the video stream *require* synchronization to be performed to the exact same frame as in the original session (see Fig. 9.11). Some user interaction requests can relax their data dependency faithfulness to video and still produce a replay faithful to the original session. By classifying user interaction requests in terms of their tolerances to frame departures, it is possible to relax the enforcement of these data dependencies. Such classification scheme is desirable.

Using the tolerance taxonomy presented in Chapter 7 for classifying temporal relationships, we classify user interaction requests in terms of the request's tolerance to video

| Interaction Request | Target Frame Precision |
|---|---|
| zooming of video | lazy |
| freezing video frame | absolute |
| increasing replay speed of video | *laissez faire* |

**Table 9.2: Relative tolerance of some typical user interaction requests over the video stream. The target frame precision specifies the minimum precision required to satisfy the correctness of such request under timing variability conditions.**

frame departures.

- absolute:

  $j = j^*$ (i.e., the interaction must be replayed *wrt* the same video frame referenced in the original session).

- lazy:

  $j - \alpha_1 < j^* < j + \alpha_2$ (i.e., the interaction can be replayed either a few frames ahead or behind the true reference).

- laissez-faire:

  $k = j^*$ (i.e., the interaction will be replayed anyway it happens, without regard to the status *wrt* the true reference).

For example, the use of this classification scheme is illustrated in Table 9.2, which shows a possible classification of some common interaction requests over the video stream.

The solution requires ensuring two conditions during replay. First, that $r_i$ equals $r_i^*$. That is, both record and replay observe the same sequence of requests, in terms of elements and ordering. Note that timing-wise, the streams may be different. Second, that $v_j$ *suits* $r_i^*$. That is, data dependencies (between user interactions and video) observed in the original session are satisfied.

To synchronize these streams, we need to accomplish the following steps. First, enforce the dependency relationship $s(r_i \leftarrow v_j^*)$. This step preserves the correctness of interactions with the video stream. Second, enforce the synchronization relationships $(A \leftarrow VW)$. This step preserves the lip synchronous relationship between audio, window, and video updates. Finally, we need to address the significant and context of the $R$ stream. Consider the addition of a new stream $R$.

Define the stream $R^*$ as the collection of data references originally observed during the capture of a session. Define the stream $R$ as the collection of data references observed during a replay of the session. Each $r_i$ *will refer* to some $v_j$. Each $r_i^*$ *refers* to some $v_j^*$. $n$ events $(r_1 \cdots r_n)$. The $R$ stream is a logging stream, i.e., a progress assertion stream. The stream is generated by specific user interactions that affect and depend on the video stream. The event generation triggers of this stream must be added to an application, so as to preserve the relationship of interactions to data. This step is needed since this relationship is created by an asynchronous request. Let's consider the user interaction request as another stream $R$ (a logging stream). A dependency constraint $d_{ij}$ introduced by some *user interaction request* $r_i$ with the video stream (at some video frame $v_j$) could be modeled as:

$$d_{ij} = (r_i \rightarrow v_j) \tag{9.3}$$

However, this approach has the drawback of unnecessary fine grain on the specification of synchronization, which will counter the results previously obtained. Another approach is to model the dependency as

$$d_{ik} = (r_i \rightarrow a_k) \tag{9.4}$$

During record, the request stream posts a $r_i$ event as a *hierarchical synchronization event* to the audio stream. During playback, the functional re-execution of the events will result in a request $r_i'$ to be generated. Also, during playback, the record request progress $r_i$ will be posted and logged. Since at playback time, the audio stream is synchronized to the request event $r_i$. By synchronizing the corresponding record and playback request events $r_i$ and $r_i'$. component between playback and

$$(r_i \leftarrow r_i') \tag{9.5}$$

## 9.7 Concluding Remarks

In this chapter, I reviewed issues on the specification and development of replayable applets. I examined the developer interfaces, provided guidelines for developers, and provided insight into the delivery of temporal-awareness to an applet.

# CHAPTER 10

# CONCLUSION

*Two roads diverged in a wood, and I*
*... I took the one less traveled by,*
*And that has made all the difference.*
— *The Road Not Taken. by Robert Frost.*

This dissertation was about computer users, the workspaces they use, and the sessions they spent on those. In the pursuit of a particular activity, sessions represented time spent while workspaces represented the tools used to produce work artifacts. Both represented work forms, but unlike workspaces, sessions lack reusability between individuals. The sessions and workspaces I studied here were instead objects or entities of specifiable behavior. In this dissertation, I discussed strategy, tools, and mechanisms to support the reuse of sessions by different users, at a later time. The goal was the design of mechanisms for the capture, storage, playback, and manipulation of workspaces as artifacts for the support of collaborative work. To this extent, I described a paradigm and its associated collaboration artifact that pioneered a new approach to computer-supported collaborative work.

The research stems from the approach chosen to playback a session: the playback of a workspace through the re-execution of inputs to its underlying computation's abstract base machines. During capture, this required recording state and inputs to applets in a workspace. During playback, it recreated the workspace spanned by these applets by scheduling and re-executing such inputs as temporal media streams. Furthermore, to increase the collaborative content of the later-time playback, the capture of the session was augmented through audio-based annotation.

The playback and manipulation of such a workspace enabled its interactive delayed-sharing or in terms of CSCW, an asynchronous-sharing of a workspace. This research enabled the modeling of a workspace as a time-dependent collaboration artifact, that unlike most authored artifacts, possessed an implicit authoring model.

A re-executable representation introduced several problems. These input streams represented a new kind of temporal media streams, referred to as re-executable content streams. I showed re-executable content streams to be significantly different from continuous media streams. They are more susceptible to variations on load and platform performance. Their events are also stateful and asynchronous. These heterogeneous characteristics strongly affected the formulation of a feasible stream synchronization solution. The use of re-executable content streams also impacted the realization of mechanisms for collaboration. I presented a flexible, media-independent, framework for the playback of workspaces. The framework provided support for heterogeneous temporal relationships through tradeoffs between synchronization and continuity monitored through the use of statistical process controls.

## 10.1 Future Work

The research of re-executable content is very young and very promising. My research provides enabling contributions to the areas of collaborative systems as well as on multimedia systems. My paradigm and session objects can re-define the way we look at the Internet. I envision some components of my research entering near-maturation stages, for example, when domain-specific models are chosen such as for interactive training systems, audio-annotated interactive workbooks, and transportable interactive demos. This vision is validated by our work on the Medical Collaboration Testbed at the University of Michigan.

The support of advanced collaborative features for asynchronous use of session objects introduces an interesting problem in their representation that naturally benefits from research on multimedia databases. Narasimhalu survey of research issues on multimedia databases [77], presented the following considerations for research on the representation problem: data model, dimensions, real-time data, representation of complex objects, and transcoded objects. In this dissertation, I presented a data model for session objects that supports the heterogeneous multimedia dimensions through media-independent modeling of the underlying temporal media streams. I show how this data model provides building blocks for the design of efficient mechanisms for the manipulation of session objects.

Integration of this media-independent data model with a real-time multimedia database seems the next logical step. Such integration should allow research on the querying and parsing of session objects so as to allow content retrieval of arbitrary sequences — in this dissertation, I discussed the use of sequences as an aid mechanism to abstract content from re-executable content streams. across a session object composed of both re-executable and continuous media streams. Whereas a multimedia database would allow for the static parsing of results of semantical queries over sequences of re-executable content; on the other hand, a real-time multimedia database would support dynamic playback of queries over such sequences.

Finally, research on transcoded objects represents a very exciting area of research. Session objects are inherently versioned objects; they associate a re-executable content artifact to a temporally-aware application. Automatic transcoding can the occur with respect to either of these: (1) transcoding to a different representation of the artifact contents; (2) transcoding to a different version of the temporally-aware application. The former relates to creating and managing (co-existing) relationships between multiple representations of a given session. The later relates to mechanisms to detect and handle obsolecensce of re-executable content due to changes on the underlying workspace intended for re-execution. Both transcoders require playback of a session object coupled with intelligent parsing of their meta representation.

## 10.2 Recapitulation

This dissertation introduced the notion of interactive delayed-sharing of a session on a computer-supported workspace to allow reuse of such session at a later time. A data artifact, referred to as a session object, was used to represent and encapsulate the session. The session object was composed of heterogeneous multimedia streams that represented temporally-ordered input sequences to applets in a workspace. Playback of such a session recreated the underlying workspace spanned by these applets through the streaming and re-execution of these input sequences in their respective applets. The contributions of this dissertation were as follows.

First, this dissertation pioneered the re-execution approach to the record and replay of sessions for the support of computer-supported collaborative work. In doing so, it introduced a novel paradigm for flexible support of asynchronous collaboration that allowed users to collaborate by annotating, editing, and refining these delayed-shared workspaces. This dissertation explored these collaborative features but focused mostly, on the record, modeling, and playback of these workspaces.

Second, this dissertation introduced the notion of workspaces as authored, transportable, and temporally-aware artifacts and investigated flexible mechanisms for the delivery of such temporal-awareness to workspaces composed of heterogeneous applications through the decoupling of tool and media services. This dissertation developed a formal model and specification of these workspaces through the use of flexible temporal relationships.

Last, this dissertation described mechanisms for the integration of re-executable content streams together with continuous multimedia streams. It described a media integration framework for flexible integration of richly heterogeneous multimedia streams (as found on these replayable workspaces). The framework provided media-independent policies for the playback of relative interval-based scheduling of fine-grained asynchronous events subject to relative timing constraints (both causal and temporal) on multiple $n$-ary relationships that are enforced through pair-wise (binary) synchronization points. The dissertation introduces the use of statistical process controls to monitor the performance of media integration through statistical relaxation and/or constraint of temporal relationships.

While the above are important contributions on by themselves, this dissertation, also, lay down a path for future research on the browsing, service provisioning, and authoring of future transportable workspaces in future ubiquitous computing environments.

APPENDICES

# APPENDIX A

# ESTIMATION OF ASYNCHRONY

In the determination of run-time synchronization treatment measures, the session manager needs an estimate of the inter-stream asynchrony for a given synchronization relationship. The session manager estimates for each pair of applet streams $(a, b)$ in the synchronization relationship a pair-wise inter-stream asynchrony estimate $(\epsilon_i^*)$, as follows. For each stream, during the processing of each synchronization checkpoint $s_i$, the session manager collects timestamps for both its (1) starting time and (2) ending time. The timestamps of $b$ and $a$ streams are used to compute two pairwise asynchrony estimates:

- just *before* the $i^{th}$ synchronization checkpoint:
  $\epsilon_{in}^*(b, a) = start_i^b - start_j^a$;
- just *after* the $i^{th}$ synchronization checkpoint:
  $\epsilon_{out}^*(b, a) = end_i^b - end_j^a$.

# APPENDIX B

# THE TIMING VARIABILITY MODEL

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

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

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 $f_o()$, the *realizable scheduling interval duration* $T$ becomes

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

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

$$T^* = E(e_1) + \Delta_{1,2} + \cdots + E(e_n) + \Delta_{n,s_1} \tag{B.3}$$

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

$$\epsilon^* = T - T^* = \sum f_o(e_i) \tag{B.4}$$

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 departure — to timing variability during replay at a workstation. Consequently, we obtain a *$\omega$-compensated scheduling interval duration* $T_\omega$ formulated as

$$T_\omega = E(e_1) + f_o(e_1) + \omega\Delta_{1,2} + \cdots E(e_n) + f_o(e_n) + \omega\Delta_{n,s_1} \tag{B.5}$$

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

$$\epsilon_\omega = \sum f_o(e_i) - (1-\omega)\sum \Delta_{i,i+1} \tag{B.6}$$

and rewritten as

$$\epsilon_\omega = \epsilon^* - (1-\omega)\sum \Delta_{i,i+1} \tag{B.7}$$

This is important because now, the replay of a $\omega$-compensated scheduling interval accounts for the estimated departure from timing faithfulness, $\epsilon^* = \sum f_o()$. Trends in asynchrony are modeled as departures from timing-wise faithfulness to the original schedule. The compensation factor $\omega$, determined at run-time, is 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.

# APPENDIX C

# SYNCHRONIZATION AND CONTINUITY

From our results in [68], both playback smoothness and continuity estimators depend on the synchronization costs $q$ as follows.

- playback smoothness of asynchronous streams;

  We propose to relate playback smoothness to how schedule time departs from elapsed real-time. For a smooth playback, we want such departures to be small and stable. Let $m_i$ be the ratio of schedule time (LTS) to real time (RT), i.e., $m_i = \frac{LTS_i - LTS_{i-1}}{RT_i - RT_{i-1}}$. Because we are dealing with re-executable content, scheduling needs to be adjusted to compensate for trends during playback on workstations of unequal performance. For example, such compensations are found on the compression and/or expansion of resources such as: (1) idle schedule time [52, 66], (2) silence [31, 45], and (3) presentation frame rate [76]. Let $w_i$ be the adaptation effort over a schedule time interval $T$, thus resulting in the compression (or expansion) of $T_i$ into a playback real-time duration $w_i T$. At the end of $T$, synchronization is performed with associated costs $q(a)$. Updating our metric $m_i$ with these values, we find that playback smoothness is affected by synchronization costs $q(a)$ as follows:

  $$m_i(a) = \frac{T}{w_i T + q(a)} \tag{C.1}$$

  The compensation effort $w_i$ reduces the impact of synchronization costs over playback smoothness. Therefore, we want, over time, a compensation effort that is smooth and effective. That is, we desire $m_i$ to exhibit exhibit controlled variability over an average value of 1.

- continuity jitter of continuous streams;

  Let $c_i$ be the timing departure from the expected duration ($c$) of a media frame to the actual real-time elapsed during its playback: $c_i = \frac{c}{RT_i - RT_{i-1}}$. At the end of the playback of the frame, synchronization is performed with associated costs $q(a)$. Updating our metric $c_i$ when $T = c$, we find that playback continuity is affected by synchronization costs as follows:

  $$c_i(a) = \frac{T}{T + q(a)} \tag{C.2}$$

  No surprise here. When synchronization costs are small, the metric $c_i$ tends to 1. When synchronization costs are large, $c_i$ is smaller than 1. When synchronization costs are variable (continuity jitter), $c_i$ exhibits strong variability over an expected value of 1. We desire the metric $c_i$ to exhibit exhibit controlled variability over an average value of 1.

Similarly, synchronization is easily related to synchronization costs, as follows. Synchronization costs $q(a)$ are dependent on the inter-stream asynchrony $\epsilon_i$. The synchronization costs $q(a)$ depend on the size of synchronization wait $w(a)$ imposed by the session manager during playback. Among other things, this waiting time $w(a)$ depends on the current inter-stream asynchrony $\epsilon_i$ and the treatment measure selected by the session manager to treat the asynchrony $\epsilon_i$.

# APPENDIX D

# PROTOCOL P3

The adaptive algorithm undergoes five phases, as shown in Fig. 4.5. 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 slave stream 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 slave stream and the last known $j^{th}$ synchronization event seen on the master stream, the $i^{th}$ inter-stream asynchrony estimate $\epsilon_i$ is generated as:
  $\epsilon_i = t(master_j) - t(slave_i)$.

- `STREAM-SYNCHRONIZATION`: If $i > j$ the slave stream waits until the $i^{th}$ synchronization event occurs on the master 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 $\rho$ = DECREASE
    if $(isWindowBehindTrend())$ then $\rho$ = INCREASE
```

$else\ \rho =$ SAME

- FORECAST-UPDATE: The replay speed for the $(i+1)^{th}$ scheduling interval is updated as:
  $if\ (\rho ==$ DECREASE$)\ then\ \omega_{i+1} = \omega_i + \gamma_i(\epsilon_i)$
  $if\ (\rho ==$ INCREASE$)\ then\ \omega_{i+1} = \omega_i - \gamma_i(\epsilon_i)$
  $if\ (\rho ==$ 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 step function,
  $$\gamma_i(\epsilon_i) = \left\{ \begin{array}{lll} 10c & for & |\epsilon_i| > 2T_1 \\ c & for & -T_1 \leq \epsilon_i \leq T_1 \end{array} \right\}$$
  where $c$ represents a fine-tuning constant over the scheduling process controls. A value of $c = 0.001$ was (empirically) found to work well for the selected streams. However, it would be interesting to investigate the optimility of a formulation of $c$ centered around probability-weights. We could use such formulation to properly update the schedule compensation based on the statistical significance of the smoothed schedule departure of $\epsilon_i$.

- POLICY-IMPLEMENTATION: For every event in the slave stream on the next $(i+1)^{th}$ scheduling interval, the inter-event delay time between slave stream events $k$ and $k+1$ is set to:
  $\Delta_{k,k+1} = \{t(slave_{k+1}) - t(slave_k)\} * \omega_{i+1}$

# APPENDIX E

# PROTOCOL P5

The relaxation algorithm undergoes five phases. The **inputs** to the algorithm are the stream tolerance to timing departures $T_1$ and the past inter-stream asynchrony history between this stream and its master, $(\epsilon_0, \cdots, \epsilon_i)$. The algorithm **outputs** the relaxation decision $SyncFire_{i+1}$.

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 slave stream and the last known $j^{th}$ synchronization event seen on the master stream, the $i^{th}$ inter-stream asynchrony estimate $\epsilon_i$ is generated as:
  $\epsilon_i = t(master_j) - t(slave_i)$.

- `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:

```
$if\ (|\epsilon_i| > T_1)\ and\ (|\epsilon_{i-1}| > T_1)\ then$
    $if\ (isSlaveAheadTrend())\ then\ \rho = \texttt{DECREASE}$
    $if\ (isSlaveBehindTrend())\ then\ \rho = \texttt{INCREASE}$
    $else\ \rho = \texttt{SAME}$
```

- `FORECAST-UPDATE`: The replay speed for the $(i+1)^{th}$ scheduling interval is updated as:

```
$if\ (\rho == \texttt{DECREASE})\ then\ SyncFire_{i+1} = YES$
```

$if\ (\rho ==$ INCREASE$)\ then\ SyncFire_{i+1} = YES$

$if\ (\rho ==$ SAME$)\ then\ SyncFire_{i+1} = NO$

- `STREAM-SYNCHRONIZATION`: If $SyncFire_{i+1}$ equals $NO$, synchronization is relaxed. Otherwise, if $i > j$ the slave stream waits until the $i^{th}$ synchronization event occurs on the master stream. Otherwise, do nothing.

- `SCHEDULE-MEASURE`: For every event in the slave stream on the next $(i+1)^{th}$ scheduling interval, the inter-event delay time between slave stream events $k$ and $k+1$ is set to:
  $\Delta_{k,k+1} = \{t(slave_{k+1}) - t(slave_k)\} * \omega_{i+1}$
  as determined by associated Policy P3.

# APPENDIX F

# SYNCHRONIZATION POLICIES

## P1 Policy: 0-way

The P1 policy relies on short-term scheduling corrections. If, during playback, a scheduling interval last less than during record, an idle time adjustment is inserted to match the current timing departure. If the scheduling interval last more, nothing is done. There is no synchronization between $a$ and $b$.

## P2 Policy: 1-way

The P2 policy relies on short-term scheduling corrections. Synchronization is 1-way between $a$ and $b$; whenever $b$ is ahead of $a$, $b$ waits for $a$.

## P3 Policy: Adaptive 1-way

The P3 policy relies on long-term scheduling corrections. Synchronization is 1-way; whenever $b$ is ahead of $a$, applet stream $b$ waits for $a$. The long-term scheduling compensation is computed during run-time, based on the statistical process performance of the relationship (see Fig. 8.5). Our scheduling controls implement long-term measures over the scheduling of a stream as follows. For a given relationship, we specify a tolerance region $(\mu_\epsilon, \sigma_\epsilon)$ over a smoothed indicator of the inter-stream asynchrony $\epsilon_i$. The tolerance region is used to characterize the compensation effort *wrt* to the asynchrony departure. The strength of the compensation effort is related to the statistical significance of the smoothed asynchrony departure.

### P4 Policy: 2-way

The P4 policy relies on short-term scheduling corrections. Synchronization is 2-way; whenever $b$ is ahead of $a$, $b$ waits for $a$ and whenever $a$ is ahead of $b$, $a$ waits for $b$.

### P5 Policy: Adaptive 2-way

The P5 policy relies on long-term scheduling corrections. Synchronization is relaxed 2-way. Whenever $b$ is ahead of $a$, $b$ may wait for $a$ and whenever $a$ is ahead of $b$, $a$ may wait for $b$. This relaxation of synchronization is based on the run-time statistical process performance of the relationship. The relaxation of synchronization firing controls is implemented as follows (see Fig. 8.6). For a given relationship, we specify a tolerance region $(\mu_\epsilon, \sigma_\epsilon)$ over a smoothed indicator of the inter-stream asynchrony $\epsilon_i$. The tolerance region is used to characterize the relaxation effort *wrt* to the statistical significance of the asynchrony departure. Blocking synchronization measures are initiated when a statistical significant trend and/or departure is detected. Adaptive scheduling measures (as specifed by the P3 policy) may be coupled with this mechanism to compensate for long-term playback performance trends.

# APPENDIX G

# WORKSPACE AND MEDIA LAYERS

During playback, the session manager provides a unified, VCR-like, view of the replayable workspace to the user. The session manager translates user requests to commands to applets on its workspace. Such applet operations are orchestrated by the session manager. The workspace API supports the orchestration of multiple applets. The workspace layer is coupled to the media layer. Workspace operations must wait for media operations to become stable. The media layer is said to be stable after inter-stream synchronization has been performed. Each applet's media layer interfaces to the session manager's media layer, which enforces various synchronization protocols between applets. The layers of the replayable workspace are shown in Fig. G.1.

The media layer is composed of two threads: (1) scheduling and (2) synchronization. The threads are mutually exclusive. The scheduling thread provides a critical section for $tms$ event re-execution. The synchronization thread provides a critical section for handling of synchronized playback of a workspace.

The media layer is composed of $n + 1$ threads: 1 update-listener() and $n$ constraint-handlers. The update-listener() thread listens for Sync-Updates() messages from stream controllers and dispatches a constraint-handler() thread to handle any Sync-Update() message it receives. Ultimately, this results in the firing of a Sync-Cont() message to the originating stream controller. The various constraint-handler() threads are mutually exclusive.

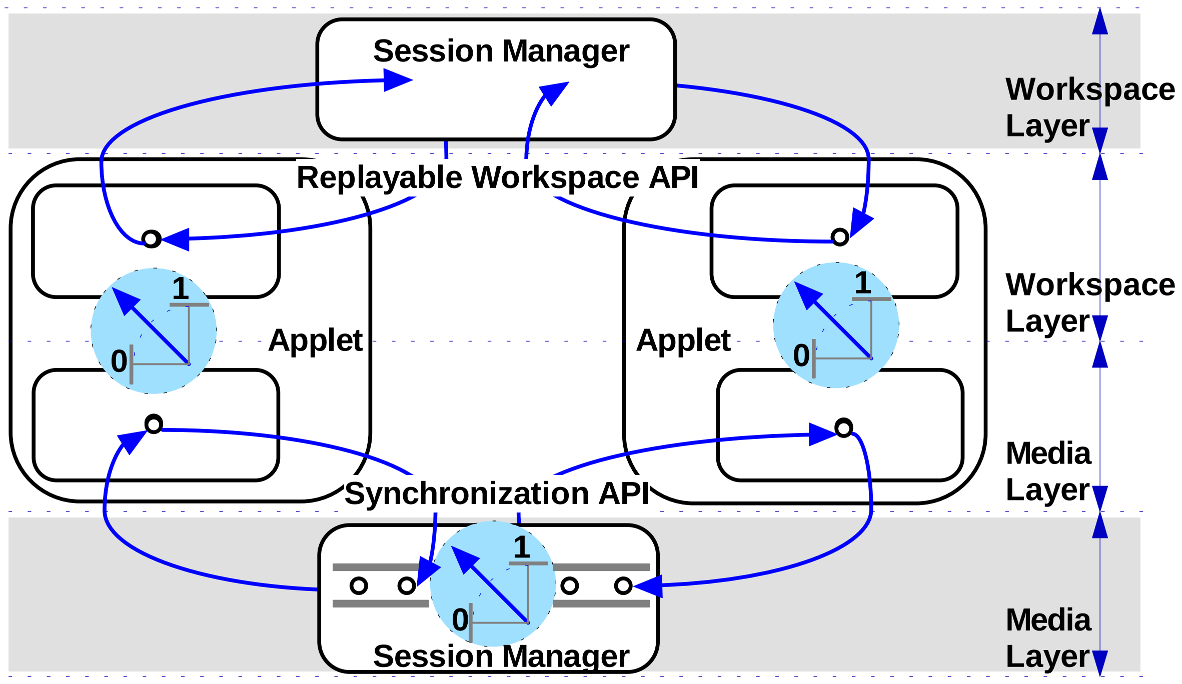


Figure G.1: A view to the decoupling of media and workspace layers on the framework. A replayable workspace is spanned by multiple replayable applets. Replayable applets are orchestrated by a session manager. A user interacts with the replayable workspace through a VCR-like interface provided by the session manager. Applets and session manager are composed of two layers: media layer and workspace layer. An applet's workspace layer is coupled to its media layer. Each applet's media layer interfaces to the session manager's media layer, which enforces synchronization between applets.

# APPENDIX H

# CONSTRAINT UPDATE RESOLUTION ALGORITHM

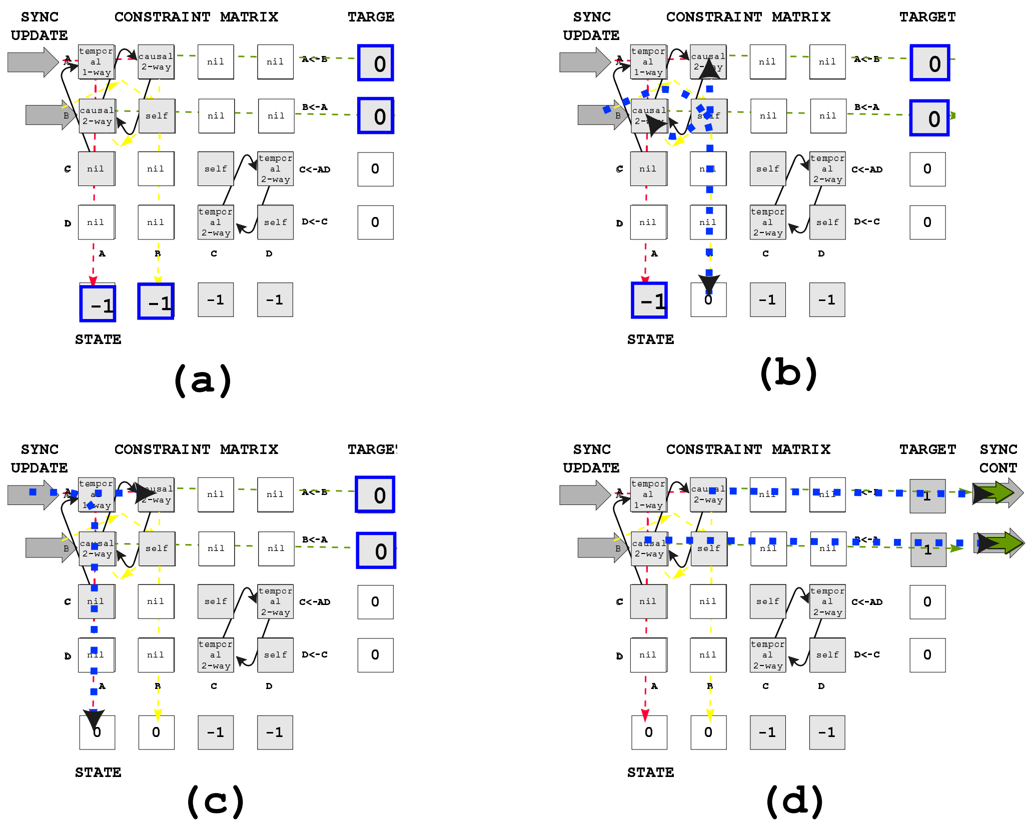


Figure H.1: The session manager updates its global constraint resolution matrix with each `Update()`. Depending on the relationships, one or more applet streams may be released from the synchronization barrier. Panels (a-d) show the handling of the tight synchronization pair $(a : b)$. Panel (a) shows the initial state, updates are about to be received at any time. In panel (b) shows an update for $b$ is received and propagated. In panel (c) shows an update for $a$ is received and propagated. In panel (d) shows the tight synchronization pair is met and streams $a$ and $b$ are released.

```
/* ************************************************************************* */
int        num_streams;
int        num_clients;
/* ************************************************************************* */


/* ************************************************************************* */
mutex_t        state_access_barrier;
thread_t    *rhs_handler[MAXSTREAMS];
thread_t    *lhs_handler[MAXSTREAMS];
cond_t        update_from[MAXSTREAMS];
/* ************************************************************************* */


/* ************************************************************************* */
main(argc, argv)
    int     argc;
    char    *argv[];
{
    char    *filename_ptr;
    int    portnumber;

    init_manager( portnumber, num_clients, filename_ptr );

    while( forever ) {
        LISTEN_FOR_SYNC_UPDATES();

        forever = WORKSPACE_LAYER_COUPLING();
    }

    clean_manager();

    return;
}
/* ************************************************************************* */
/* *********************************************************************** */
void
LISTEN_FOR_SYNC_UPDATES( ) {
    sync_msg = GET_NEXT_SYNC_UPDATE();

    MUTEX_LOCK( &state_access_barrier );
        j = GET_SENDER( sync_msg );
        update[j].lts_in = LTS();
        update[j].lts_out = LTS();
        update[j].wait = update[j].lts_out - update[j].lts_in;
        update[j].async = 0;
        UPDATE_CONSTRAINTS_WITH( str );
    MUTEX_UNLOCK( &state_access_barrier );

    SPAN_THREAD_TO(handle_update_from, j, rhs_handler[j]);
}
/* *********************************************************************** */


/* *********************************************************************** */
void
UPDATE_CONSTRAINTS_WITH( j ) {
    state[ j ] = state[ j ] + 1;
    BROADCAST_UPDATE( j );
    return;
}
/* *********************************************************************** */


/* *********************************************************************** */
void *
HANDLE_UPDATE_FROM( j ) {
    MUTEX_LOCK( &state_access_barrier );
        LHS_CONSTRAINT_HANDLER( j );
    MUTEX_UNLOCK( &state_access_barrier );
    thr_exit();
}
/* ************************************************************************** */


/* ************************************************************************** */
void
LHS_CONSTRAINT_HANDLER( j ) {
    int         i;
    int         start, end;

    MODLOOP_SETUP(start, end, i, num_clients);
    MODLOOP_START(i, end)
        APPLY_PAIRWISE_FULFILLMENT( j , i );
    MODLOOP_END(i, end, num_clients)

    ISSUE_SYNC_CONT_FOR( i );

    return;
}
/* ************************************************************************** */


/* ************************************************************************** */
void
ISSUE_SYNC_CONT_FOR( j ) {
    update[j].lts_out = LTS();
    update[j].wait = update[j].lts_out - update[j].lts_in;

    target[j] = target[j] + 1;

    update[j].async_case = RHS_SAME_LHS;
    update[j].adapt_rate = nil;
    update[j].adapt_flag = FALSE;

    DISPATCH_SYNC_CONT( j , target[j], update[j] );
    return;
}
/* ************************************************************************ */


/* ************************************************************************ */
int
APPLY_PAIRWISE_FULFILLMENT( i, j ) {
    int         dependency;
    int         async_case;
    int         treatment;
    ACCURATE     lts_diff;

    dependency = CONSTRAINT( i, j );
    if( dependency == -1 ) return;

    async_case = DPC( i, j );
    if( async_case == RHS_SAME_LHS ) return;

    treatment = LOOKUP_TREATMENT_FOR( dependency, async_case );
    APPLY_SYNC_TREATMENT( i, j, treatment, async_case );

    return;
}
/* ************************************************************************ */



/* ************************************************************************ */
void
APPLY_SYNC_TREATMENT( i, j, treatment, async_case) {
    switch( treatment ) {
    case SYNC_NOTHING:
        SYNC_NOTHING( rhs );
        break;
    case SYNC_WAIT_ADAPT:
        if( a_case == RHS_AHEAD_LHS )
            SYNC_ADAPT( lhs, rhs, SPEED_DECREASE );
        else
            SYNC_ADAPT( lhs, rhs, SPEED_INCREASE );

    case SYNC_WAIT:
        if (DPC(lhs, rhs) != RHS_SAME_LHS)
            if( a_case == RHS_AHEAD_LHS )
                SYNC_WAIT( lhs );
            else
                SYNC_WAIT( rhs );
        break;

    case SYNC_ADAPT:
        if( a_case == RHS_AHEAD_LHS )
            SYNC_ADAPT( lhs, rhs, SPEED_DECREASE );
        else
            SYNC_ADAPT( lhs, rhs, SPEED_INCREASE );
        break;
    }
    return;
}
/* ************************************************************************** */


/* ************************************************************************** */
UPDATE_MEAN( async_in, sync_num, lhs, rhs ) {
    ACCURATE t, old_mean, new_mean;

    old_mean = GET_ASYNC_MEAN_FOR( lhs, rhs );
    new_mean = async_in;

    if( sync_num >= INTERVENTION_DELAY ) {
        t = (old_mean * (ACCURATE) (sync_number-1)) + async_in;
        new_mean = t / (ACCURATE) sync_num;
    }

    SET_ASYNC_MEAN_FOR( lhs, rhs, new_mean);
    return new_mean;
}

DPC(lhs, rhs) {
    ACCURATE    async_in;
    SYNC_NUM    sync_num;

    if( GET_CONSTRAINT(lhs, rhs) == P0 )
        return RHS_NOT_RELATED;

    switch( GET_CONSTRAINT(lhs, rhs) ) {
    case P0:
    case P1:
    case P2:
    case P4:
        if( GET_STATE(lhs) > GET_STATE(rhs) )
            return RHS_BEHIND_LHS;

        if( GET_STATE(lhs) < GET_STATE(rhs) )
            return RHS_AHEAD_LHS;

        return RHS_SAME_LHS;

    case P3:
    case P5:
    default:
        async_in = asynchrony_log[lhs][sync_num].lts_in -
                   asynchrony_log[rhs][sync_num].lts_in;
        mu_async = UPDATE_MEAN( async_in, sync_num, lhs, rhs );

        if( fabs(async_in) > (mu_async  + ASYNC_TOLERANCE))
            SYNC_ADAPT( lhs, rhs, counter_measure );
            SYNC_WAIT( str );
        else
            SYNC_ADAPT( lhs, rhs, counter_measure );
    }
}


int
LOOKUP_TREATMENT_FOR( protocol, async_case ) {
    int         treatment;

    switch( protocol ) {
    case P1:
        switch( async_case ) {
        case RHS_BEHIND_LHS:
            treatment = SYNC_NOTHING;
            break;
        case RHS_SAME_LHS:
            treatment = SYNC_NOTHING;
            break;
        case RHS_AHEAD_LHS:
            treatment = SYNC_NOTHING;
            break;
        }
        break;
    case P2:
        switch( async_case ) {
        case RHS_BEHIND_LHS:
            treatment = SYNC_WAIT;
            break;
        case RHS_SAME_LHS:
            treatment = SYNC_NOTHING;
            break;
        case RHS_AHEAD_LHS:
            treatment = SYNC_NOTHING;
            break;
    }
    break;
case P3:
    switch( async_case ) {
    case RHS_BEHIND_LHS:
        treatment = SYNC_WAIT;
        break;
    case RHS_SAME_LHS:
        treatment = SYNC_NOTHING;
        break;
    case RHS_AHEAD_LHS:
        treatment = SYNC_ADAPT;
        break;
    }
    break;
case P4:
    switch( async_case ) {
    case RHS_BEHIND_LHS:
        treatment = SYNC_WAIT;
        break;
    case RHS_SAME_LHS:
        treatment = SYNC_NOTHING;
        break;
    case RHS_AHEAD_LHS:
        treatment = SYNC_WAIT;
        break;
    }
    break;
case P5:
    switch( async_case ) {
    case RHS_BEHIND_LHS:
        treatment = SYNC_WAIT_ADAPT;
        break;
    case RHS_SAME_LHS:
        treatment = SYNC_NOTHING;
        break;
        case RHS_AHEAD_LHS:
            treatment = SYNC_WAIT_ADAPT;
            break;
        }
        break;
    }
    return treatment;
}
/* ************************************************************************ */


/* ************************************************************************ */
void
SYNC_NOTHING( str ) {
    /* Should allow sync_cont to flow in all cases */
    return;
}


void
SYNC_WAIT( str ) {
    WAIT_FOR_UPDATE_ON( str );
    return;
}


SYNC_ADAPT( lhs, rhs, counter_measure ) {
    update[j].adapt_flag = counter_measure;
    adapt_rate = COMPUTE_NEW_ADAPT_RATE( lhs, rhs );
    update[j].adapt_rate = adapt_rate;
    return;
}
/* ************************************************************************ */


/* ************************************************************************ */
```

```
BROADCAST_UPDATE( i ) {
    COND_BROADCAST( &update_from[i] );
    return;
}


WAIT_FOR_UPDATE_ON( STREAM_ID i ) {
    COND_WAIT( &update_from[i], &state_access_barrier );
    return;
}
/* ************************************************************************ */


INIT_MANAGER() {
    startTime = getTimeStamp();
    MUTEX_INIT( &state_access_barrier );
    INIT_STATE( filename_ptr );
    return;
}


void
INIT_CONSTRAINTS( char * filename ) {
    STREAM_ID    rhs, lhs;
    int        protocol;
    FILE        *fp = NULL;


    fp = fopen( filename, "r" );

    ...

    for( lhs=0; lhs<num_streams; lhs++)
        for( rhs=0; rhs<num_streams; rhs++) {
            fscanf( fp, "%d", &protocol);
            SET_CONSTRAINT(lhs, rhs, protocol);
        }

    fclose( fp );
    return;
}


void
init_state( char * filename_ptr ) {
    STREAM_ID    str;
    ACCURATE    init_time;

    init_constraints( filename_ptr );
    for( str=0; str<num_streams; str++) {
        set_constraint_target(str, 0);
        set_STATE(str, -1);
    }

    init_time = LTS();
    for( str=0; str<MAXSTREAMS; str++ ) {
        update[str].async_case = RHS_SAME_LHS;
        update[str].adapt_flag = FALSE;
        update[str].adapt_rate = nil;
        update[str].start_time = 0.0;
        update[str].end_time = 0.0;
    }

    for( str=0; str<MAXSTREAMS; str++ )
        COND_INIT( &update_from[str] );

    return;
}


void
clean_manager() {
    MUTEX_FREE( &state_access_barrier );
    clean_server_socket();
    return;
}
/* ************************************************************************* */
```

# BIBLIOGRAPHY

# BIBLIOGRAPHY

[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 Comm. 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] F. Arman, R. Depommier, A. Hsu, and M.Y. Chiu. Content-based browsing of video sequences. In *Proc. of ACM Multimedia '94*, pages 173–180, San Francisco, CA, USA, October 1994.

[4] 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.

[5] P.C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *Transactions on Computer Systems*, 13(1):1–31, 1995.

[6] John Bazik. XMX: Copyright 1988, 1989, 1990, Brown University, Dept, Providence, RI, USA.

[7] P.A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, February 1996.

[8] K. Birman et al. *The ISIS System Manual, Version 2.0*, April 1990.

[9] K.P. Birman and T.A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, pages 47–76, February 1987.

[10] J. Birnbaum. How the coming digital utility may reshape computing and telecommunications. IEEE Media Briefing, October 1996.

[11] J. Birnbaum. Pervasive information systems. *Communications of the ACM*, 40(2):40–41, February 1997.

[12] L. Brothers, V. Sembugamoorthy, and M. Muller. ICICLE: Groupware for code inspection. In *Proc. of the Second Conference on Computer-Supported Cooperative Work*, pages 169–181, October 1990.

[13] 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.

[14] 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.

[15] 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.

[16] M.S. Chen, D.D. Kandlur, and P.S. Yu. Support for fully interactive playout in a disk-array-based video server. In *Proc. of ACM Multimedia '94*, pages 391–398, San Francisco, CA, USA, October 1994.

[17] G. Chung, K. Jeffay, and H. Adbel-Wahab. Dynamic participation in computer-based conferencing system. *Journal of Computer Communications*, 17(1):7–16, January 1994.

[18] C.R. Clauer, J.D. Kelly, T.J. Rosenberg, C.E. Rasmussen, P. Stauning, E. Friis-Christensen, R.J. Niciejewski, T.L. Killeen, S.B. Mende, Y. Zambre, T.E. Weymouth, A. Prakash, G.M. Olson S.E. McDaniel, T.A. Finholt, and D.E. Atkins. A new project to support scientific collaboration electronically. *EOS Transactions on American Geophysical Union*, 75, June 28 1994.

[19] R. Clauer, J.D. Kelly, T.J. Rosenberg, C.E.P. Stauning, E. Friis-Christensen, R.J. Niciejewski, T.L. Killeen, S. Mende, T.E. Weymouth, A. Prakash, S.E. McDaniel, G.M. Olsen, T.A. Finholt, and D.E. Atkins. UARC: A Prototype Upper Atmostpheric Research Collaboratory. *EOS Trans. American Geophys. Union*, 267(74), 1993.

[20] E.J. Conklin. gIBIS: A hypertext tool for exploratory policy discussion. In *Proc. Groupware '92*, pages 133–137, 1992.

[21] J. Conklin and M.L. Begeman. gIBIS: A hypertext tool for exploratory policy discussion. In *Proc. of the Second Conference on Computer-Supported Cooperative Work*, pages 140–152, October 1988.

[22] J.J. Courant. Softbench message connector: Customizing software development tool interactions. *HP Journal*, 45(4):34–39, June 1994.

[23] 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.

[24] 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.

[25] C. Crowley. TkReplay: Record and Replay for Tk. In *USENIX Tcl/Tk Workshop*, pages 131–140, Toronto, Canada, July 1996.

[26] 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, 1 1993.

[27] P. Dewan. Flexible user interface coupling in collaborative systems. In *Proc. otf the ACM CHI'91 Conference on Human Factors in Computing Systems*, pages 41–48, April 1991.

[28] P. Dewan and R. Choudhary. Primitives for programming multi-user interfaces. In *Proc. of the Fourth ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 69–78, November 1991.

[29] P. Dewan and R. Choudhary. A flexible and high-level framework for implementing multi-user user interfaces. *ACM Transactions on Information Systems*, 10(4):345–380, October 1992.

[30] M. Diaz and P Senac. Time stream petri nets: A model for multimedia streams synchronization. In *Proc. of the First Internation Conference on Multi Media Modeling*, pages 257–274, Singapore, November 1993.

[31] 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.

[32] C. Ellis and J. Wainer. A conceptual model of groupware. In *Proc. of the Fifth Conference on Computer-Supported Cooperative Work*, pages 79–88, Raleigh, N.C., USA, November 1994.

[33] R. Fish, R. Kraut, M. Leland, and M. Cohen. Quilt: A collaborative tool for cooperative writing. In *Proc. of ACM SIGOIS Conference*, pages 30–37, 1988.

[34] T. Fisher. Real-time scheduling support in ultrix-4.2 for multimedia communication. 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.

[35] S. Flinn. Coordinating heterogeneous time-based media between independent applications. In *Proc. of ACM Multimedia '95*, pages 435–444, San Francisco, CA, November 1995.

[36] G.E. Fry, A. Jourdan, D.Y. Lee, A.S. Sawkar, N.J. Shah, and W.C. Wiberg. Next generation wireless networks. *Bell Labs Technical Journal*, 1(2):88–96, Autumn 1996.

[37] J.D. Gabbe, A. Ginsberg, and B.S. Robinson. Towards intelligent recognition of multimedia episodes in real-time applications. In *Proc. of ACM Multimedia '94*, pages 227–236, San Francisco, CA, USA, October 1994.

[38] D. Garfinkel, B.C. Welti, and T.W. Yip. HP SharedX: A tool for real-time collaboration. *HP Journal*, 45(4):26–33, April 1994.

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

[40] 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.

[41] V. Goldberg, M. Safran, and E. Shapiro. Active Mail: A framework for implementing groupware. In *Proc. of the Fourth Conference on Computer-Supported Cooperative Work*, pages 75–83, Toronto, Canada, October 1992.

[42] J.L. Herlocker and J.A. Konstan. Commands as media: design and implementation of a command stream. In *Proc. of ACM Multimedia '95*, pages 155–166, San Francisco, CA, November 1995.

[43] T.Y. Hou, A. Hsu, M.Y. Chiu, S.K. Chang, and H.J. Chang. An active multimedia system for delayed conferencing. *Proceedings of the SPIE, High-speed Networking and Multimedia Computing; SPIE*, 2188:97–104, 1994.

[44] K. Jeffay. On latency management in time-shared operating systems. In *Proc. of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 86–90, Seattle, WA, USA, May 1994.

[45] 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.

[46] K. Jeffay and D.L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proc. of the 14th IEEE Real-Time Systems Symposium*, pages 212–221, Raleigh-Durham, NC, USA, December 1993.

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

[48] D.R. Jefferson. Synchronization in distributed simulation. *Proc. International Computers in Engineering 4-8 Aug. 1985, publ by ASME, New York, NY, USA*, pages 407–413, 1985.

[49] O. Jones. Multidisplay software in x: A survey of architectures. *The X Resource, O'Reilly & Associates*, June 1993.

[50] S. Kaplan, W. Tolone, D. Bogia, and C. Bignoli. Flexible, active support for collaborative work with ConversationBuilder. In *Proc. of the Fourth Conference on Computer-Supported Cooperative Work*, pages 378–385, Toronto, Canada, October 1992.

[51] 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.

[52] M.Y. Kim and J. Song. Multimedia documents with elastic time. In *Proc. of ACM Multimedia '95*, pages 143–154, San Francisco, CA, November 1995.

[53] L. Kleinrock. Nomadicity: Anytime, anywhere in a disconnected world. *Mobile Networks and Applications*, 1(4), January 1996.

[54] H.F. Korth and A. Silbershatz. Database research faces the information explosion. *Communications of the ACM*, 40(2):139–142, February 1997.

[55] K.Y. Lai, T.W. Malone, and K.C. Yu. Object Lens: A "spreadsheet" for cooperative work. *ACM Transactions on Office and Information Systems*, 6(4):332–353, 1988.

[56] B. Leiner, V. Cerf, D. Clark, R. Kahn, L. Kleinrock, D. Lynch, J. Postel, L. Roberts, and S. Wolf. The past and future history of the internet. *Communications of the ACM*, 40(2):102–108, February 1997.

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

[58] L. Li, A. Karmouch, and N.D. Georganas. Multimedia teleorchestra with independent sources: Part 1 — temporal modeling of collaborative multimedia scenarios. *Multimedia Systems*, 1(2):143–153, Spring 1994.

[59] Y. Ligier, O. Ratib, M. Logean, and C. Girard. Osiris : A medical image manipulation system. *M.D. Computing Journal*, 11(4):212–218, July-August 94.

[60] C.J. Lindblad, D.J. Wetherall, and D.L. Tennenhouse. The VuSystem: A programming system for visual processing of digital video. In *Proc. of ACM Multimedia '94*, pages 307–314, San Francisco, CA, USA, October 1994.

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

[62] T.D.C. Little. A framework for synchronous delivery of time-dependent multimedia systems. *Multimedia Systems*, 1(1):87–94, 1993.

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

[64] T. Malone, K.R. Grant, F.A. Turbak, S.A. Brobst, and M.D. Cohen. Intelligent information sharing systems. *Communications of the ACM*, 30(5):390–402, May 1987.

[65] 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.

[66] N.R. Manohar and A. Prakash. Dealing with synchronization and timing variability in the playback of interactive session recordings. In *Proc. of ACM Multimedia '95*, pages 45–56, San Francisco, CA, November 1995.

[67] 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*, pages 161–177, Stockholm, Sweden, September 1995.

[68] N.R. Manohar and A. Prakash. Design issues on heterogeneous replayable workspaces. Technical Report Technical Report CSE-, Department of Eletrical Engineering and Computer Science, University of Michigan, 1996.

[69] N.R. Manohar and A. Prakash. A flexible architecture for heterogeneous media integration on replayable workspaces. In *Proc. Third IEEE Int'l Conf on Multimedia Computing and Systems, to appear*, Hiroshima, Japan, June 1996.

[70] N.R. Manohar and A. Prakash. Flexible tool coordination and media integration on heterogeneous replayable workspaces. Technical Report Technical Report CSE-, Department of Eletrical Engineering and Computer Science, University of Michigan, 1996.

[71] 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.

[72] C.W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proceedings of the IEEE ICMCS*, May 1994.

[73] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1):39–65, 1986.

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

[75] 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.

[76] T. Nakajima and H. Tezuka. A Continuous Media Application supporting Dynamic QoS Control on Real Time Mach. In *Proc. of ACM Multimedia '94*, pages 289–297, San Francisco, CA, USA, October 1994.

[77] A.D. Narasimhalu. Multimedia databases. *Multimedia Systems*, 4(5):226–249, October 1996.

[78] C.M. Neuwirth, D.S. Kaufer, R. Chandhok, and J.H. Morris. Issues in the design of computer support for co-authoring and commenting. In *Proc. of the Third Conference on Computer-Supported Cooperative Work*, pages 183–195, Los Angeles, CA, USA, October 1990.

[79] C.M. Neuwirth, D.S. Kaufer, J. Morris, and R. Chandhok. Flexible diff-ing in a collaborative writing system. In *Proc. of the Fourth Conference on Computer-Supported Cooperative Work*, pages 147–154, Toronto, Canada, October 1992.

[80] S.R. Newcomb, N.A. Kipp, and V.T. Newcomb. The Hytime hypermedia/time-based document structuring language. *Communications of the ACM*, 34(11):67–83, November 1991.

[81] B. Ozden, R. Rastogi, and A. Silverschatz. A framework for the storage and retrieval of continuous media data. In *Proc. of the IEEE Int'l Conference on Multimedia Computing and Systems*, pages 580–589?, May 1995.

[82] D.L. Parnas and D.P. Siewiorek. Use of the concept of transparency in the design of hierarchically structured systems. *Communications of the ACM*, 18(7):401–408, 7 1975.

[83] A. Prakash and H.S. Shim. Distview: Support for building efficient collaborative applications using replicated objects. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work (CSCW'94)*, pages 153–164, Chapel Hill, N.C., October 1994.

[84] R. Rajkumar. *Real-Time Synchronization in Uniprocessors*, pages 15–58. Kluwer Academic Publishing, 1991.

[85] S. Ramanathan and P. Venkat Rangan. Continuous media synchronization in distributed multimedia systems. In *Proc. of the 3rd Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, pages 328–335, La Jolla, CA, USA, November 1992.

[86] S. Ramanathan and Venkat P. Rangan. Adaptive feedback techniques for synchronized multimedia retrieval over integrated networks. *ieanep*, 1(2):246–260, April 1993.

[87] P. Venkat Rangan and Harrick M. Vin. Designing file systems for digital audio and video. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 81–94. Association for Computing Machinery SIGOPS, October 1991.

[88] M. Roseman and S. Greenberg. GroupKit: A groupware toolkit for building real-time conferencing applications. In *Proc. of the Fourth Conference on Computer-Supported Cooperative Work*, pages 43–50, Toronto, Canada, October 1992.

[89] 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.

[90] 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.

[91] D.C. Schmidt, M. Fayad, and R.E. Johnson. Software patterns: Guest editor notes. *Communications of the ACM*, 39(10):36–39, October 1996.

[92] E.M. Schooler. Conferencing and collaborative computing. *Multimedia Systems*, 4(5):210–225, October 1996.

[93] H. Schulzrinne. Operating system issues for continuous media. *Multimedia Systems*, 4(5):269–280, October 1996.

[94] National Research Council (Computer Science, Telecommunications Board Commission on Physical Sciences Mathematics, and Applications). National Collaboratories: Applying Information Technology for Scientific Research. National Academy Press, 1993.

[95] A. Sheperd, N. Mayer, and A. Kuchinsky. Strudel: An extensible electronic conversation toolkit. In *Proc. of the Second Conference on Computer-Supported Cooperative Work*, October 1990.

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

[97] R. Steinmetz. Analyzing the multimedia operating system. *IEEE MultiMedia*, 2(1):68–84, Spring 1995.

[98] R. Steinmetz and K. Nahrstedt. *Chapter 15: Synchronization*, pages 585–595. Prentice Hall, 1995.

[99] D.L. Stone and K. Jeffay. An empirical study of delay jitter management policies. *ACM Multimedia Systems*, 2(6):267–279, January 1995.

[100] N.A. Streitz, J. Geiler, J.M. Haake, and J. Hol. DOLPHIN: Integrated meeting support across liveboards, local and remote desktop environments. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work (CSCW'94)*, pages 345–358., Chapel Hill, N.C., October 1994.

[101] B.I. Szabo and G.K. Wallace. Design considerations for JPEG video and synchronized audion in a Unix workstation environment. In *Proc. of the Summer USENIX Conference*, pages 353–367, Nashville, TN, USA, Summer 1991.

[102] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[103] H.M. Vin, P.T. Zellweger, D.C. Swinehart, and P. Venkat Rangan. Multimedia conferencing in the etherphone environment. *Computer*, 24(10):69–8, October 1991.

[104] Marc Weiser. Some computer science issues in ubiquitous computing. *CACM*, 36(7):74–84, 1993.