Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work,
September 10–14, Stockholm, Sweden
H. Marmolin, Y. Sundblad, and K. Schmidt (Editors)

# The Session Capture and Replay Paradigm for Asynchronous Collaboration

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

In this paper, we describe a paradigm and its associated collaboration artifact to allow flexible support for asynchronous collaboration Under this paradigm, a user session with an application's user interface is encapsulated into a data artifact, referred to as a *session object* Users collaborate by annotating, by modifying, and by a back-and-forth exchange of these session objects Each session object is composed of several data streams that encapsulate audio annotations and user interactions with the application The replay of a session object is accomplished by dispatching these data streams to the application for re-execution Re-execution of these streams is kept synchronized to maintain faithfulness to the original recording The basic mechanisms allow a participant who misses a session with an application to catch up on the activities that occurred during the session This paper presents the paradigm, its applications, its design, and our preliminary experience with its use

## Introduction

Many approaches to computer supported collaboration have been centered around synchronous collaboration [5, 6, 10]. In synchronous collaboration, users of a multi-user application first find a common time and then work in a WYSIWIS (What You See Is What I See) collaborative session. However, a synchronous mode of collaboration can often be too imposing on the schedule of the participants. It requires that users be able to find a common time to work together but, in many cases, that is not easy.

Several systems for the support of asynchronous collaboration provide ways to model the interactions among users and the evolution of collaboration repositories [8, 11, 15]. In this paper, we present a complimentary paradigm for

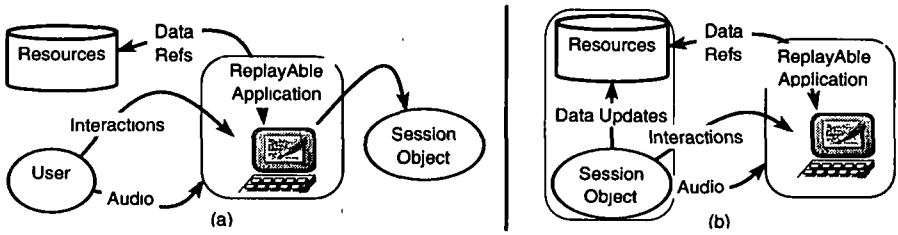


Figure 1 Capture and replay of an interactive session

asynchronous collaboration that allows users to record and replay an interactive session with an application. We refer to this paradigm as **WYSNIWIST** (*What You See Now Is What I Saw Then*) [12]. The paradigm introduces an associated data artifact, the *session object*, used to capture the collaborative session. Figure 1 shows a high level view of the capture and replay of an interactive session with an application. During the capture of the session, user interactions with the application, audio annotations, and resource references (e.g., fonts, files, environment) are recorded into a session object (Figure 1a). The replay of the session uses the data stored in the session object to recreate the look and feel of the original session (Figure 1b).

Our replay approach is application-dependent. During replay, input events are re-executed (as opposed to a passive replay form such as a series of screen dumps). The re-execution approach to session replay exhibits benefits that are of particular interest in collaborative work. 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 paper is organized as follows. First, we illustrate some applications of the paradigm. Next, we present the goals of our design. Then, we discuss related work. Next, we describe how session objects are modeled. Then, we discuss the design issues in building a system to support the paradigm. Next, we describe the implementation of a prototype. Finally, we discuss our experiences with its use and present some concluding remarks.

# Examples

The next examples illustrate how different asynchronous collaboration scenarios could benefit from both the paradigm and its artifact.

### Using the paradigm to support synchronous collaboration

Our work was originally motivated by the UARC[1] project, a collaboratory experiment among domain scientists in a wide-area network [4]. 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 *apriori* 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.

### 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 [2]. 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.

## 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

[1] Upper Atmospheric Research Collaboratory

streams and to the need to maintain a consistent view of referenced resource inputs.

- 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.

Finally, the following conditions are assumed to maintain the determinism of the replay: the application performs deterministic computations; the state and events that affect the computation can be captured; and the same application is used for record and replay.

## Related Work

Our work is related to work in screen capture, and pseudo-servers, collaboration-aware systems, conversation systems, and multimedia replay systems.

**Screen recorders**, such as WATCHME (for NEXTs), work at the screen level, intercepting and recording screen updates (or even screen dumps). Updates from *all* applications are captured, as opposed to those from a specific application. Because these approaches maintain the external look of the screen, interaction with the underlying application or its artifacts is not possible.

**X pseudo-servers** such as SHAREDX and XTV [1, 3] intercept events sent by applications being shared to the window server. These systems are primarily targeted for synchronous work. However, if these events are recorded and the state of the window server captured, this approach, in principle, allows the events to be replayed for specific applications. While our approach does not need to reconstruct the interface, these approaches must reconstruct it using low-level updates. Furthermore, replay is limited to only those events that go through the window server.

Our approach is based on the **exchange and refinement** of a work-in-progress by group members. Several other systems also support this paradigm, such as CONVERSATIONBUILDER [9] and STRUDEL [16]. These systems work by first defining a shared object and then formalizing a protocol that defines and limits the transactions that modify this object. In both cases, the shared object is an argumentative tree. In these systems, interactions are usually in reference to some data artifacts. Our paradigm introduces a new data artifact, which can be used to enrich intra-task descriptions on these systems.

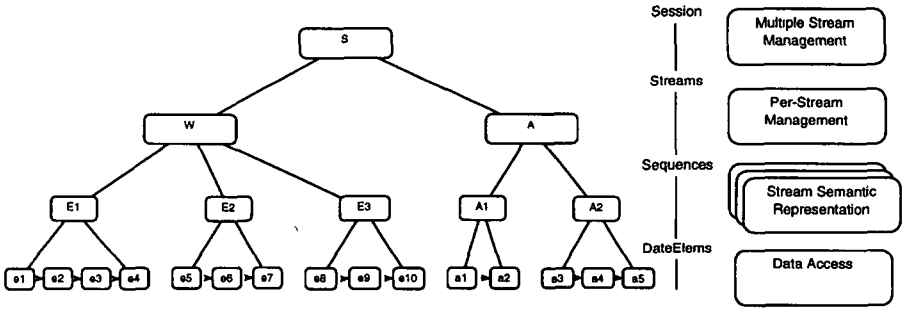


Figure 2 The object hierarchy (a) and corresponding abstraction layers (b)

**Collaborative writing systems** such as QUILT [7] and PREP [14] support the idea of allowing users to work asynchronously through annotations added to a document. Our approach builds on those ideas but, in our case, the artifact being annotated is a software-based recording of interactions with an application. Annotations can be made by gesturing (e.g., telepointing), text, and audio.

## 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. Figure 2 shows the correspondence of the abstraction layers and the object hierarchy.

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 distinguish between two classes of streams: *annotation* and *functional* streams. An annotation stream contains annotation events. A functional stream updates the state of the application. The window stream (W) is typically a functional stream, whereas the audio stream is usually an annotation 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 $E_1 = \left( \texttt{MouseDown MouseUp} \right)$, 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.

# 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.

## 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

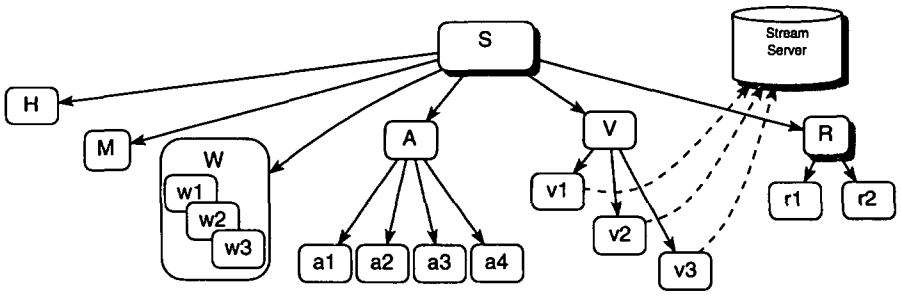


Figure 3. Storage representation of recorded sessions.

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.

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 $t$.

## 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. 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.

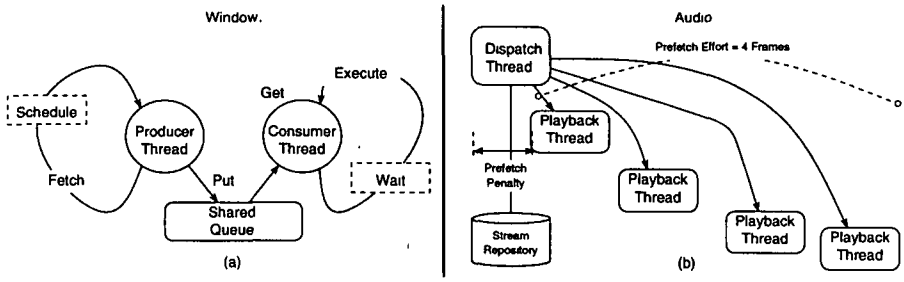


Figure 4 Thread models for replay of the window event stream (a) and for the replay of the continuous audio stream (b).

## 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 [13] showed that an adaptive protocol that attempts to compensate for varying load generally performs better across all load conditions.

Streams execute as cooperating thread tasks in a single CPU. The infrastructure provides two generic thread models. Figure 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 event replay. Figure 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 [3]. 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.

## 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 is a difficult one and remains open. We are currently exploring 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 apart the state

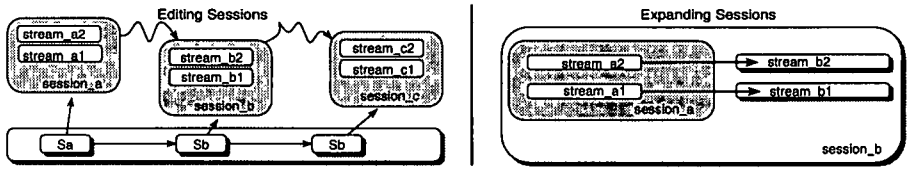


Figure 5. Editing (a) and expanding (b) session objects.

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 5 shows two of the potential ways of editing sessions. Figure 5a models editing of a session through concatenation of previously recorded session segments, $s_a, s_b$, and $s_c$. Figure 5b models a session $s_b$ that extends a previously recorded session $s_a$ with additional interactions and annotations.

## 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.

### 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).

### 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 $S$ (recall example in Fig. 3) reduces to the problem of mailing of directories. 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.

## Implementation

We implemented an object-oriented prototype toolkit for NEXT workstations under the Mach Operating System. The toolkit provides the REPLAYABLE object class. The REPLAYABLE class provides applications with transparent access to the infrastructure services. A MACDRAW-like object oriented drawing application and a text editor application were retrofitted with the toolkit. REPLAYABLE applications access the paradigm features through menus and windows added to the application. The session controller window provided by the prototype to each REPLAYABLE application is shown in Fig. 6. The infrastructure allows applications to: (1) re-execute window events (e.g., ges-

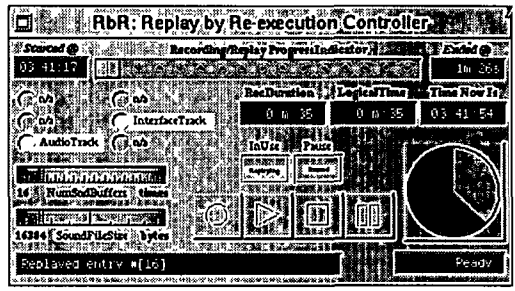


Figure 6 View of the REPLAYABLE application controller.

turing, typing, moving windows), (2) record voice-annotations, (3) provide synchronized replay of these streams, and (4) to replay selected streams.

The infrastructure provides a *logical time system* (LTS) to support time-stamping of events. It also provides efficient, disk-based, read and write of streams. Finally, it provides per-stream scheduling and inter-stream synchronization protocols to support faithful replay of streams.

The prototype currently supports two streams: a *discrete stream* (i.e., window events) and a *continuous stream* (i.e., audio). Each stream is dispatched to a separate processor. The window event stream is dispatched to the CPU — which is subject to arbitrary load conditions. The audio stream is dispatched to the DSP — assumed to be dedicated. These components (application, streams, DSP, CPU, infrastructure services, disk, and data paths) are shown in Fig. 7. Side (a) shows the record-time view and side (b) shows the replay-time view of the prototype.

We designed an adaptive synchronization protocol that attempts to: (1) maintain statistical control over inter-stream asynchrony, and (2) update a weighted history forecast formulation to determine the presence of a significant trend in the asynchrony history. The adaptive behavior of the protocol works as follows. If the current asynchrony is large enough, the past asynchrony history is examined to determine the presence of a trend in the asynchrony. If such a trend exists, the window stream schedule is either compressed or expanded — thus increasing or decreasing the relative replay speed of the window stream, respectively.

## Experience

Our experience with the prototype shows several results The duration of most recorded sessions tends to be of the order of a few minutes, typically 2 to 4 minutes in our experience. To capture a session of about 4 minutes, the size of our compressed document artifact was only $1.7MB$. Note that capturing the

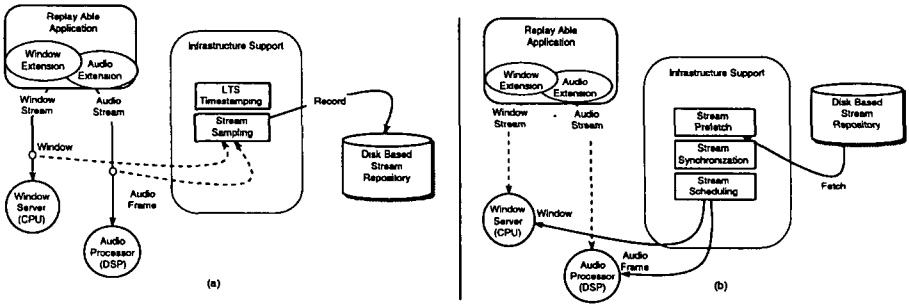


Figure 7. Application model for record (a) and replay (b) of window and audio streams.

| document artifact | artifact size (MBytes) | fidelity of replay |
|---|---|---|
| digital audio and video | 16 MBytes (4 fps, MPEG compressed) | ($1024x1024 \rightarrow 320x240$) pixels, lossy |
| session object | 1.7 MBytes | ($1024x1024 \rightarrow 1024x1024$) pixels, lossless |

Table I. Comparison of data artifacts for session capture and replay The session objects document artifact has small size and lossless fidelity of replay.

same session using digital video and audio required at least $16MB$ of storage. These results are summarized in Table I.

While users tolerated some inter-stream asynchrony — up to 1 or even 2 seconds was acceptable — most users did not tolerate audio discontinuities. We found that our adaptive synchronization protocol [13] provided acceptable performance across all load conditions. The average asynchrony $\mu_{async}$ for this protocol was about half the mean duration of the scheduling interval used. The standard deviation of the asynchrony $\sigma_{async}$ for the protocol was also about one third the mean scheduling interval duration. 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. Both parameters $\mu_{async}$ and $\sigma_{async}$ were stable across all load conditions.

Maintaining data as separate streams was a good design choice for record, storage, and access. Both streams had substantially different characteristics. The window stream had a 10 : 1 compression ratio. The stream had a variable event density between 10 to 30 events per second. For the audio stream it was possible to achieve a 2 : 1 compression ratio but only by the use of the audio-specific compression software. We found that to handle access overheads for the audio stream, it was best to amortize the overhead over several frames.

The access overheads of the prototype averaged about 2%.

There are basically three important parameters that determine access overheads for the audio stream: (1) the audio frame size; (2) the number of audio frames that are written to disk at one time during recording; and (3) the number of audio frames that are prefetched from disk at one time during playback. We summarize the appropriate values found for these parameters next. Audio frame sizes of 16 *KB* were found to work best on our platform. Since the NEXTs use threads to dispatch audio frames, using smaller frame sizes made thread scheduling overheads appreciable, so as to affect the quality of audio playback. Using much larger values did not cause significant additional improvements in performance. Writing 2 frames at a time (32 KB of data) to the disk at a time led to the best amortization of the disk penalty hit during recordings. Prefetching 4 frames at a time (64 KB of data) was found to give the best amortization of the disk penalty hit during playback.

Experience with the prototype also made us aware of the potential for the following further uses of the paradigm.

## Using the paradigm to capture collaboration content

In a recently funded NSF project, we plan to support the type of collaboration that occurs between a radiologist and a doctor over radiographs to diagnose a patient's medical problem. 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.

## 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.

### 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.

## Conclusions

In this paper, we 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-base of existing collaboration systems, so as to capture intra-task descriptions in both asynchronous and synchronous collaborative systems.

Our goals for the near future are (1) to support efficient recordings of sessions that involve interactions with image and video artifacts and (2) to use the paradigm to support asynchronous collaboration among radiologists and doctors at the University of Michigan hospitals.

## Acknowledgements

We would like to thank all the UARC project members, in particular, Amit Mathur and Hyong Shim This work has been supported in part by the University of Michigan's Rackham Merit Fellowship and by the National Science Foundation under the Grant ECS-94-22701 and under the Cooperative Agreement IRI-9216848.

## References

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

[2] L Brothers, V. Sembugamoorthy, and M Muller ICICLE· Groupware for code inspection. In *Proceedings of the Second Conference on Computer-Supported Cooperative Work*, pages 169–181, October 1990.

[3] 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.

[4] 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

[5] C Ellis, S.J. Gibbs, and G Rein. Design and use of a group editor In G. Cockton, editor, *Engineering for Human-Computer Interaction*, pages 13–25 North-Holland, Amsterdam, September 1988

[6] C.A Ellis, S J. Gibbs, and G L Rein. Groupware. Some issues and experiences. *Communications of the ACM*, pages 38–51, January 1991

[7] R. Fish, R Kraut, M Leland, and M. Cohen Quilt: A collaborative tool for cooperative writing In *Proceedings of ACM SIGOIS Conference*, pages 30–37, 1988.

[8] V Goldberg, M. Safran, and E. Shapiro. Active Mail. A framework for implementing groupware. In *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work*, pages 75–83, Toronto, Canada, October 1992

[9] S Kaplan, W. Tolone, D. Bogia, and C. Bignoli Flexible, active support for collaborative work with ConversationBuilder. In *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work*, pages 378–385, Toronto, Canada, October 1992

[10] M Knister and A. Prakash DistEdit. A distributed toolkit for supporting multiple group editors. In *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, pages 343–355, Los Angeles, California, October 1990

[11] 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

[12] 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

[13] N R. Manohar and A. Prakash Dealing with timing variability in the playback of interactive session recordings. In *submitted to· Proceedings of ACM Multimedia '95*, San Francisco, CA, USA, November 1995

[14] 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 *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, pages 183–195, Los Angeles, California, October 1990

[15] C M Neuwirth, D S Kaufer, J. Morris, and R Chandhok Flexible diff-ing in a collaborative writing system In *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work*, pages 147–154, Toronto, Canada, October 1992

[16] A. Sheperd, N Mayer, and A Kuchinsky. Strudel. An extensible electronic conversation toolkit. In *Proceedings of the Second Conference on Computer-Supported Cooperative Work*, October 1990