Replay by Re-execution: a paradigm for asynchronous collaboration via record and replay of interactive multimedia sessions

POSITION PAPER

Nelson R. Manohar and Atul Prakash
Department of Electrical Engineering and Computer Science
University of Michigan at Ann Arbor, Ann Arbor, MI 48109-2122
Phone: (313) 763-1585. Email: aprakash, nelsonr@eecs.umich.edu

AUGUST 1994

INTRODUCTION

Many approaches to computer-supported collaboration have been centered around synchronous collaboration. In synchronous collaboration, users of a multi-user application first find a common time to collaborate and then work in a WYSIWIS (What You See Is What I See) collaborative environment. Synchronous collaboration is a very powerful paradigm, but can sometimes be too imposing on the schedule of the participants, since finding common time to work together, in many cases, is not easy.

On the other hand, on-going research projects on asynchronous collaboration provide with ways to model the evolution of collaboration repositories. However, while for some application domains, the evolution of the collaboration data (i.e., *what was done*) is sufficient to capture the collaboration effort, for other domains, it is the process (i.e., *how it was done*) that carry the real collaboration effort. We refer to these how-it-was-done process instances as WYSNIWIST — (What You See Now, Is What I Saw Then) — **session recordings**. During the recording of a session, input events are recorded. During the replay of a session, input events are faithfully re-executed. This **Replay by Re-execution** paradigm exhibits benefits that are tailored for collaboration:

(1) because only input events are recorded, recorded sessions are small in size and thus easier to exchange among collaborators,

(2) because input events are re-executed, the re-execution approach allows

for interactive replay (i.e., small interactions *within* a recorded session).

Session recordings are represented using temporal multimedia streams. However, unlike other temporal multimedia stream models, synchronization can be specified between continuous media (e.g., audio) and discrete media (e.g., window events, commands). Therefore, our paradigm needs to be supported by a synchronization model that compensates for performance differences between record and replay workstations.

An object-oriented prototype of this system model has been implemented in Objective-C for NeXT workstations. The prototype system provides primitives that allow applications to: (1) allow re-execution of user's actions over the user interface (e.g., gesturing, button's callbacks, typing, moving windows), (2) allow user to record voice-annotations, and (3) allow the synchronized re-execution of these streams.

AN EXAMPLE

To illustrate the use of this paradigm consider the asynchronous collaboration 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. In order for this collaboration to be effective, this collaboration needs to be augmented by reference to images. In particular, radiologist and doctor need to be able to construct a *live* 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 one or more images, and add text or graphical annotations. They then need to be able to save, send, archive, retrieve, and replay such sessions. Such digital, high resolution session recordings will not only help capture radiologists' *diagnostic conclusions*, but also their *diagnostic process*. This is important because in many instances how the diagnosis was reached is more important than the diagnosis itself.

The paradigm extends the work in asynchronous collaboration by facilitating the exchange of multimedia session recordings amongst radiologists and physicians. These recordings must be faithfully replayed on playback. For example, while examining an ultrasound scan, a radiologist may do the following actions, some of them simultaneously:

- make verbal remarks;

- point or gesture about specific regions in the scan with a mouse;
- freeze, zoom-in/zoom-out, invoke digital enhancement operations to better show a region, and rewind/advance the scan;
- graphically annotate an interesting region of a frame (image); and
- bring up on the screen other data/images/scan for comparison purposes.

By capturing the process of how a diagnosis was reached we hope to efficiently enhance the collaboration between radiologist and doctor, while making best use of their time and schedules.

ISSUES

Several system level issues must be addressed to support such a paradigm. The paradigm supports several types of streams:

- **continuous streams:** For example, temporal multimedia streams such as audio and video.
- **interface stream:** An interface stream contains user interface events. Although the interface stream is conceptually similar to a functional stream, the interface stream operates only on the state of the user interface.
- **functional streams:** A functional stream contains functions or operations (e.g., callbacks). Therefore, its replayer is a function that applies functions over the state of an application. The purpose of the functional stream is to update the computational state of the application.
- **data streams:** A data stream contains self-contained input instances, which are integral units of computation.

All streams/tracks are *implicitly related by time* to the underlying baseline recording. However, users should not be required to replay all tracks at one time. Indeed, hardware constraints (such as only one audio device) may require that users select only one track of a particular type for playback. Dependencies between tracks may also make it unnecessary to playback certain tracks (e.g., output tracks) if the information on them can be derived from other tracks (e.g., input tracks).

The issues that must be addressed include the following:

(1) mechanisms for recording the above type of sessions, which involves both continuous and discrete media, and saving the recording into file(s);

(2) mechanisms for mailing such session recordings efficiently over a network;

(3) mechanisms for faithful playback of the recording at the receiver's workstation, with the receiver having control over the playback;

(4) mechanisms to allow a back-and-forth exchange of messages that include session recordings and refer to earlier recordings; and

(5) an architecture of the system with the above mechanisms.

Our proposed work differs from the earlier work in that the earlier work has largely focussed on issue (3) — how to synchronize multiple media streams, usually assumed to be continuous media. Similarly to continuous multimedia servers, instead of handling network latency variations, the protocols would have to handle variations in access times to read streams from files/external storage. However, issues (1), (2), and (4) also need to be addressed because these recordings can get large and, if mechanisms for the above are not designed carefully, might swamp the workstations' disk space, operating system's communication buffers, and possibly the network.

DESIGN APPROACH

A prototype implementation of this system model has been implemented in Objective-C on the NeXT platform. The implementation consists of four separate components:

- **Session Manager** — provides interface to the user and manages multiple session objects;
- **Application Extensions** — for each stream, a record and replay module is provided;
- **Record Manager** — provides time stamping and persistency services to all streams; and
- **Replay Manager** — implements synchronization protocols and formulates scheduling policies across streams.

Currently, the Application Extensions consists of primitives for recording and replaying audio and interface streams. The functionality of the prototype allows applications using its primitives to record, replay and (audibly) annotate sessions with the application's GUI. Whenever, a new recording is created, events are sampled from each stream and recorded into the corresponding track of the recording. Events are time-stamped and ordered within a track. At replay time, events are scheduled and dispatched to the appropriate event handler for that track. Recorded sessions can be saved, exchanged and replayed later on by users.

Asynchronous collaboration requires *support to allows users to exchange, modify and build upon recorded sessions.* The paradigm views recordings as database objects, each composed of orthogonally defined tracks. A track is the data representation of a sampled stream. Furthermore, to keep the recorded session size small and thus support collaboration exchanges, references to data are either **by-copy** or **by-reference**, depending on the size of the data being handled and the extent of sharing across different recordings — for example, interface events are just copied, while audio and image frames are handled by reference.

In order to support the paradigm, *efficient I/O handling of streams is needed.* The prototype implements efficient per-stream I/O handling. To efficiently support recording of a stream, I/O parameters for write blocking and memory buffering can be adjusted to reduce the sampling overhead — i.e., due to file system latencies — imposed onto the stream. Similarly, to efficiently support replaying of a track, I/O parameters for prefetch effort and memory buffering can be adjusted to fit synchronization tolerances of each track.

During replay, *re-execution of tracks must be kept synchronized.* Several track synchronization algorithms have been implemented:

- **no synchronization** — tracks are only scheduled, synchronization of tracks events is not considered;

- **1-way blocking** — synchronization event causes events of a slave track to wait for its master track;

- **1-way adaptive** — as above, however, further scheduling of slave tracks is also adapted to reflect estimate processor performance;

- **2-way blocking** — synchronization event causes either track (slave or master) to wait for the other one, when out of synchronization departure exceeds tolerance levels; and
- **2-way adaptive** — as above, however, further scheduling of slave tracks is also adapted to reflect estimate processor performance.

Synchronization guarantees need to be provided by the synchronization algorithm to match application requirements. Synchronization guarantees can be either statistical, fixed or both. For example, our 2-way adaptive policy provides fixed guarantees for maximum asynchrony departures from master and slave track schedules while providing σ-based statistical control guidelines for determining asynchrony trends to then, adapt – if necessary – future scheduling of slave tracks events. An adaptive strategy has been implemented to compress or expand logical time based on indirect estimates of processor performance for a given track.

Of these, the best performance is obtained from the 2-way adaptive algorithm. This algorithm trades off occasional sound discontinuities — due to the re-synching of the master stream — to stream synchronization. The 1-way adaptive algorithm yielded the best synchronization performance of any of the 1-way algorithms. As other 1-way algorithms, the 1-way adaptive algorithm does not sacrifice sound continuity. However, unlike 2-way algorithms, the 1-way adaptive can not guarantee asynchrony departures under every load condition. Furthermore, under variable load conditions its susceptibility to the accuracy of the forecast formulation affects the quality of the performance estimators.

CONCLUSION

In this paper, we presented a new paradigm for asynchronous collaboration and its underlying synchronization algorithm classes that allow users to record and re-execute interactive sessions with an application. Recorded sessions under the paradigm are represented by multiple temporal streams, which are kept synchronized during the replay of the session. The Replay by Re-execution paradigm exhibits properties that do not restrict its replay capability to just the user interface, but more importantly, it exhibits potential collaboration benefits that outweighs its implementation cost to applications.