

## Internet-based Desktops in Tcl/Tk: Collaborative and Recordable

Amit Khetawat      Hemang Lavana      Franc Brglez  
CBL (Collaborative Benchmarking Lab), Dept. of Computer Science, Box 7550  
NC State University, Raleigh, NC 27695, USA  
<http://www.cbl.ncsu.edu/>

### Abstract

*This paper addresses issues that arise when a peer group, distributed across several time zones, uses the Internet to configure and execute distributed desktop-based applications and tasks. The paper provides solutions and Tcl/Tk implementations to support (1) peer-to-peer communication/control of distributed software and computing resources over the Internet; (2) recording and playback of interactive execution of Tcl/Tk applications and collaborative sessions.*

*The summary of 540 Internet-based experiments, each relying on RecordTaker and PlaybackMaker to record, playback, and execute ReubenDesktop configurations from local, cross-state, and cross-country servers, demonstrates the effectiveness of the proposed concepts and implementation.*

**Keywords:** collaborative desktops, recording, playback, workflows, Internet.

### 1 Introduction

The Internet and the on-going evolution of the world-wide web is expected to evolve into a network without technologic, geographic or time barriers – a network over which partners, customers and employees can collaborate at any time, from anywhere, with anyone. Issues of collaboration arise when a peer group, distributed in time and space, uses the Internet to iteratively configure and execute distributed desktop-based applications and tasks. Existing systems, based on Tcl/Tk [1, 2, 3], that are available today include the the Group-kit [4, 5] for collaboration and the TkReplay [6] for recording.

Requirements that motivated this research resulted in collaborative and recording Tcl/Tk architectures and implementations that are different from the ones introduced earlier in [4, 5, 6]. Specifically, the Internet-based environment *ReubenDesktop*, prototyped in Tcl/Tk and Expect [7], leverages a novel multi-user collaborative desktop architecture that

This research was supported by contracts from the Semiconductor Research Corporation (94-DJ-553), SEMATECH (94-DJ-800), and DARPA/ARO (P-3316-EL/DAAH04-94-G-2080 and DAAG55-97-1-0345).

also supports multi-cast visualization of workflows, encapsulating distributed data, tools and communication protocols. A hierarchy of such workflows can be re-configured and re-executed by a team of collaborating users, since the *generic architecture* supports (1) effective channels of communication among peers, and (2) distributed control of applications and tasks. Data and tools encapsulated in such workflows reflect the needs of a specific peer group. The workflows described in this paper are representative of ones that may be used in the design of experiments or as an integral part of a distributed microsystems design effort [8, 9, 10, 11, 12].

In addition, we introduce a novel architecture that allows us to *record and playback* any Tcl/Tk application, including peer interactions during the *distributed and collaborative* sessions of *ReubenDesktop*.

The paper is organized into the following sections: (2) background and motivation, introducing a simple example of collaborative and recordable desktop environment; (3) collaborative Internet-based desktop environment, user view of *FlowSynchronizer* and its implementation; (4) session recording and playback environment, user view of *RecordTaker* and *PlaybackMaker* and their implementation; (5) a summary of 540 Internet-based experiments; (6) software availability and status; (7) conclusions and future work.

### 2 Background and Motivation

The *ReubenDesktop*, described in this paper as recordable and executable upon playback, satisfies the following properties as a collaborative desktop environment [9]:

**P1:** desktop is shared and multi-cast, so that each participant can observe desktop actions of others;

**P2:** desktop supports a shared and segmented ‘talk window’, so each participant can type messages to all others in his/her own window segment;

**P3:** the shared and segmented ‘talk window’ supports a token passing mechanism, so that at any time, only a *single user* controls the desktop, but can pass the token to any other user when requested.

An example of a *ReubenDesktop* satisfying properties **P1–P3** is shown in Figure 1(a). The instance of the particular desktop has been multi-cast by student Alice to her instructor Bob with a request for on-line assistance. In the case shown, the desktop consists of two windows: (1) a sample workflow that is not executing, hence the problem, and (2) a *FlowSynchronizer* window that allows Alice and Bob to ‘talk’ and describe the problem and a solution.

Here, instructor Bob could have requested and received permission from Alice to edit the workflow and thus show a solution. Instead, Bob remembers that earlier, he *recorded* a solution to a similar problem for another student. Subsequently, he decides to *playback* the pre-recorded solution, shown in Figure 1(b). By passing control to Alice (the respective *FlowSynchronizer* window is not shown), Alice can now study the solution by re-executing the *PlaybackMaker*.

It is clear that the paradigm described in this example applies to a number of situations, including design reviews, with high potential to reduce design errors or catch them early in the process, thereby significantly enhancing the productivity of the team effort.

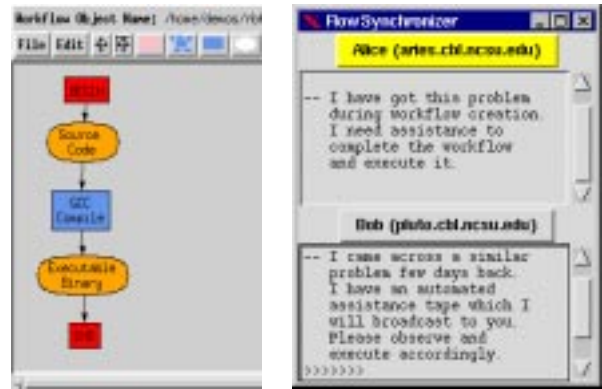
### 3 Collaborative Internet-based Desktop Environment

The Internet-based desktop environment as defined in [8] contains a number of application icons (program nodes) as well as a number of data icons (file nodes). In contrast to typical desktops of today, where data icons may be dragged and dropped onto application icons for execution, this environment allows

- *user-defined and reconfigurable execution sequences* by creating dependency edges between program nodes (application icons) and file nodes (data icons);
- *data-dependent execution sequences* by dynamic scheduling of path as well as loop executions;
- *host-transparency* as to the location of applications and data (both can reside on any host with a unique IP address).

**User View.** We use a simple example of an *Archivist Workflow* to illustrate the concepts of a collaborative Internet-based desktop environment as defined in this paper. The *ReubenDesktop* in Figure 2(a) is a snapshot of a collaborative session in progress. Two archival specialists, User1 at Host1 and User2 at Host2, are completing the tasks of archiving distributed directories to a central location (Host1). The session was initi-

(a) Collaborative description of a problem



(b) Collaborative playback of a tutorial workflow

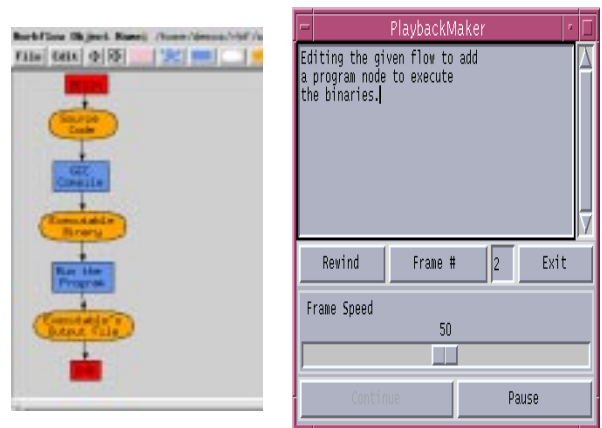
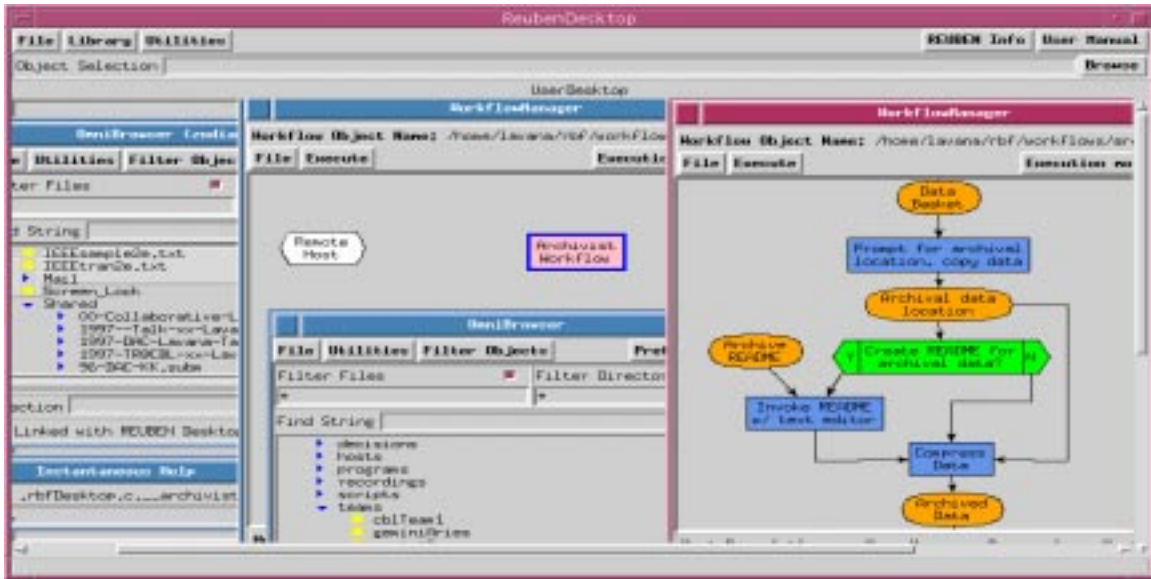


Fig. 1. Example of a collaborative remote assistance.

ated by User1. Upon invocation of the *ReubenDesktop*, User1 gained access to the UserDesktop window within the *ReubenDesktop* and the *OmniBrowser*. Using *OmniBrowser*, she selected the *TeamDefinition* configuration file, and with a single click, she initiated the session with User2. At this point both receive an identical view of the *ReubenDesktop*, along with the *FlowSynchronizer* in Figure 2(b). User1 selects the *Archivist Workflow* in the *OmniBrowser* which is now broadcast as two nodes in the new workflow window within the desktop: ‘Remote Host’ and a hierarchical workflow node named *Archivist Workflow*. This node will execute whenever any of the files selected in the *OmniBrowser* window is ‘dropped’ onto it.

As noted in the *FlowSynchronizer*, User1 initiates the archival process, by selecting directory ‘teams’ and passes the control to User2. At this point, User2 clicks on the ‘Remote Host’ which invokes another *OmniBrowser*. He selects a set of files in the ‘Shared’ directory and passes back the control. User1, by clicking on the *Archivist Workflow* node, ‘drops’ all

(a) Archivist Workflow within ReubenDesktop



(b) Archivist FlowSynchronizer



(c) About FlowSynchronizer

The *FlowSynchronizer* can support  $n$  collaborating sites, providing controlled access to two window segments at each site: (a) a token button designating the *UserSite*, and (b) a message box which provides a real-time conferencing environment. At any time, one and only one site is designated as a *Token-Holder*, by coloring its *UserSite* button in a color different from all other sites. At any time, each collaborator can transmit text in the message box to all other sites. However, only the *Token-Holder* has the capability to click on another *UserSite* button to pass the token, and hence the control of the entire environment, including any application and data displayed in the workflow window.

Fig. 2. Internet-based collaborative desktop environment.

files selected by *both* users onto this node, initiating the execution of the workflow – now shown fully expanded in the *ReubenDesktop*.

In this example, the Internet-based desktop environment has been rendered collaborative by the addition of a *FlowSynchronizer* that provides both the communication channels between collaborating participants and a control passing mechanism, supporting and maintaining the properties **P1–P3** summarized earlier. The functionality of the *FlowSynchronizer* is explained in Figure 2(c).

**Collaborative Desktop Architecture.** Two architectural extremes are possible to support collaborative activities:

1. *Replicated software architecture* - exact copies or replicas of the application being shared must be installed and maintained on each host. The application on each host handles the user interaction locally and changes made to the application state are broadcast to all other replicas to maintain the consistency of the data and the user views. GroupKit [4, 5] is an example of Tcl/Tk extension that follows this

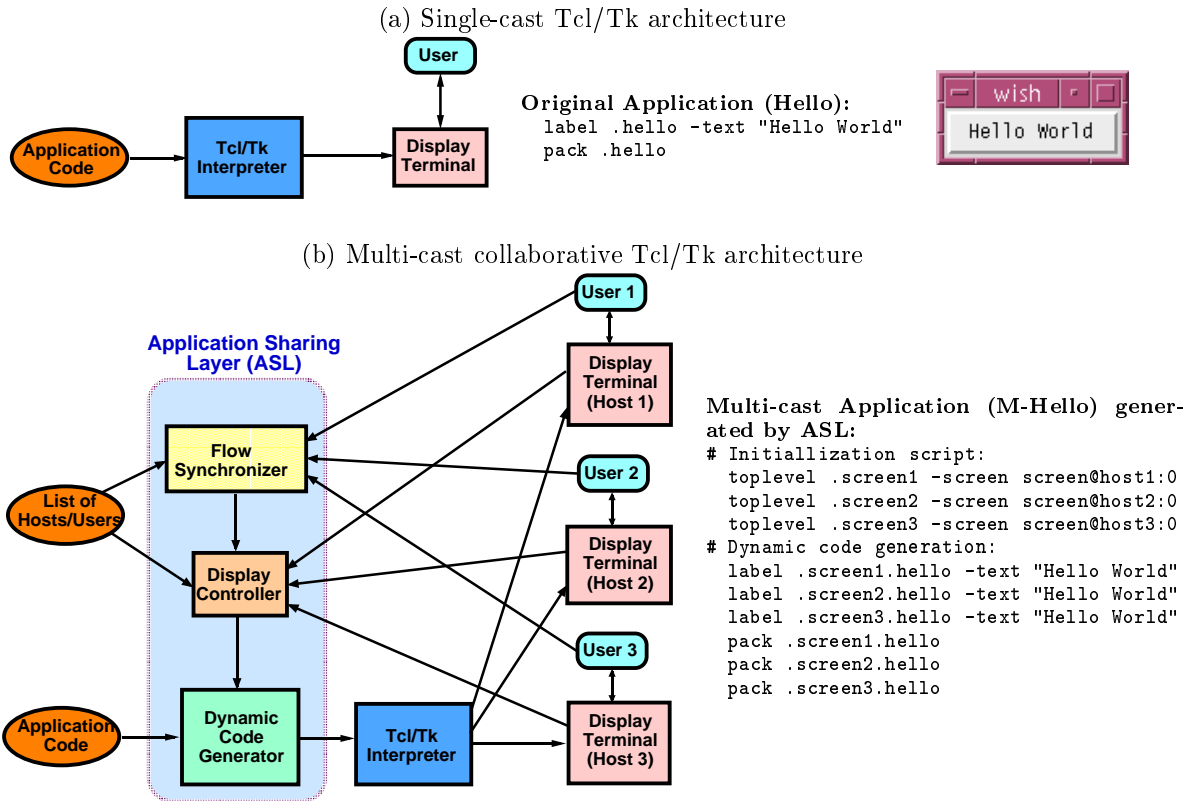


Fig. 3. Multi-user *collaborative* desktop architecture.

architecture and allows the development of Groupware applications such as drawing tools, editors and meeting tools. The major disadvantage of replicated architecture is the difficulty in keeping the data and user views consistent and in synchronization.

2. *Centralized software architecture* - a single host is responsible for setting up the collaborative session with the participating hosts which do not maintain any copies of the application being shared. Their local machines are nothing more than graphical terminals running X windows (on any UNIX, PC, or MAC workstation). The host starting the session handles all the incoming events in the form of user interactions and is responsible for sending and updating the shared applications views on all other participating hosts. The primary advantage of the centralized approach is its simplicity, since maintenance of the consistency of the application is related to a single host only.

We use *tcl-only* code to implement the centralized approach. In single-cast mode of operation shown in Figure 3(a), the application code is passed onto the Tcl/Tk interpreter. The interpreter then makes appropriate function calls to display the *graphical user interface* (GUI) on the users display terminal. For multi-casting such a GUI to ‘*n*’ display screens,

we modify the approach in Figure 3(a) to create a multi-cast collaborative mode of operation, by inserting an *Application Sharing Layer* (ASL) between the Tcl/Tk application code and the Tcl/Tk Interpreter, as shown in Figure 3(b). The ASL takes the original application code and a list of hosts as input. This layer consists of three components:

1. *FlowSynchronizer* - It provides the ability for real-time conferencing among the collaborating users.
2. *Display Controller* - It facilitates and manages ownership and exchanges of control over the application’s GUI among the collaborating users.
3. *Dynamic Code Generator* (DCG) - This provides the ability to broadcast the application GUI to ‘*n*’ displays and also process events generated from the collaborators’ displays.

**Control Mechanisms.** When a GUI of an application is broadcast to ‘*n*’ displays, it is necessary to coordinate user interaction with the application. If all the users try to interact with the application simultaneously, chaos and confusion is likely to follow soon. In most applications, however, it is essential that at any time, only one user has a control over the execution of the application. There are several mechanisms to avoid such contentions, which

include: (1) single user interaction, with other users observing, (2) round-robin based transfer of control, allowing each user to interact for a limited time only, (3) request-based control, with users deciding as to who is to interact with the application at any instant, etc.

We have implemented *request-based control* mechanism where a single user is initialized with control over the entire application that is launched. The user who has the control and can interact with the application is said to hold the *token* and is called the *Token Holder*. At any instant, the Token Holder can relinquish the control over the application by passing the *token* to any other user. The Display Controller, shown in Figure 3(b), implements this scheme by: (1) accepting and responding to user interactions such as mouse movements, keyboard events, etc., from the Token Holder only, and (2) blocking all events that are generated by any other user.

**FlowSynchronizer.** Once all the users have a shared view of the application, they still need a mechanism to communicate among themselves for effective collaboration. If a user needs to request the control of the application from the current Token Holder, she has to communicate the request to the Token Holder. We provide an additional window on each users display, called the *FlowSynchronizer* shown in Figure 2(b), which allows the users to communicate, and dynamically exchange control among themselves. It has two components for each participating site:

1. A *button* containing the name of the user as well as the host name to indicate who holds the token. The Token Holder is identified by highlighting its button by a green color whereas all other users have yellow buttons.

2. A *message box* for each user facilitates easy communication by allowing all users to simultaneously type in their respective message boxes. There are no possibilities of contentions here. This is similar to the *talk utility* available on Unix systems.

Thus, in our collaborative environment: (1) all users are allowed to interact with the message boxes for communication, and (2) only one user, the Token Holder, is allowed to interact with the entire application and relinquish the control by passing the token. We now explain some of the implementation details about the Display Controller and the FlowSynchronizer window.

*Using Grab Command to Provide Control.* Each participating user's interaction with the application is first limited to her own message box by using the Tcl command `grab`. Only the Token Holder's display screen does not have any `grab` command act-

ing on any part of its application GUI. Then, as the token is passed among different users of the collaborating team, we dynamically change the effect of `grab` acting on each of the user's screen as follows: (1) release the `grab` for the message box of the user who is being given the control, (2) achieve a `grab` for the message box of the user who is giving up the control, and (3) retain the effect of `grab` on the message boxes of all the remaining users.

*Achieving Concurrent Messaging.* It is necessary for concurrent messaging that all users be able to type into message boxes simultaneously. Because of the default class bindings provided by Tk, a user has to bring the message box into focus before she can start typing. However, if a message box is brought into focus by clicking on it, it results in message boxes of other users to lose their focus, thereby re-directing their typed message into the new message box that is now in focus.

To overcome this problem, we first disable the default class binding for the message box and add a new binding script which automatically brings the message box into focus whenever the user's mouse cursor enters it. This ensures that every user has her message box in focus whenever their mouse cursor is over it.

**Dynamic Code Generator (DCG).** The functionality of the DCG is to intercept every Tcl/Tk command of the application code and modify it to generate a new Tcl code such that it will multi-cast the GUI of the application to '*n*' displays. We use a simple application `Hello World` to illustrate the implementation details of the DCG.

Figure 3(a) shows the original application code that creates and displays a label widget containing the text "Hello World" on the user's screen. The tcl code for multi-casting such a simple example to three display screens is shown in Figure 3(b). We first create a toplevel window on each user's display screen which acts as a place holder for the various widgets that will be created by the application code. The initialization script shows how to create such place holders on each display screen using the command `toplevel`. Next, for every command in the original application code that is related to GUI, the DCG generates three commands, one for each display screen. Thus, we have three `label` and three `pack` commands in the multi-casting example code. This code merely illustrates one method to broadcast the application GUI to three display screens and does not include scripts to initialize the FlowSynchronization window and its request-based control mechanism.

## 4 Session Recording and Playback Environment

Recording and playback environment provides a mechanism of ‘taking minutes’, not only of the interactive discussions, but also of the menu-based commands associated with different tools in the workflow, of user-entered data inputs, and of user-queried data outputs. There are several benefits to the recording and playback mechanism:

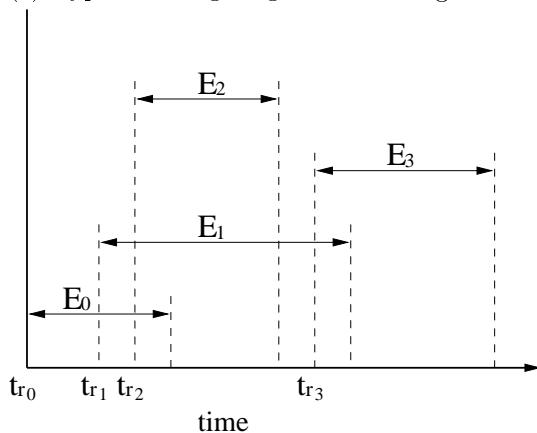
- support for automated software documentation and tutorials, capturing the dynamics of software interactions for playback and review at a later time;
- study of activities and feedback on how teams actually collaborate, to improve the effectiveness and efficiency of collaborative environments;
- remote assistance, by selecting and playing back effective solutions recorded earlier;
- recording of solutions to frequently asked questions (FAQs) - these recordings can be played back at a later time on demand;
- automated performance evaluation of tcl applications, by repeating executions of pre-recorded sessions under different cpu loads and varying conditions of network traffic - such an example is described in the next section on experiments.

TkReplay [6] is Tcl/Tk program that allows one to record all user actions of Tk programs and then replay them. It uses the Tk command `send` to intercept each user action and saving it as a script file. This script file may then be reloaded and played back later to emulate the original user actions. The TkReplay will not work with Tcl/Tk applications in which the Tk command `send` is disabled. The `send` command is potentially a serious security loophole, since any application that can connect to your X server can send scripts to your applications. These incoming scripts can use Tcl to read and write your files and invoke subprocesses under your name. Hence, it is common for applications written in Tcl/Tk to disable the the operation of the `send` command. Our desktop environment is an example of one such Tcl/Tk application in which the `send` command has been disabled. Therefore, TkReplay is not suitable for use within our desktop. Instead, we make use of the Tk command `event generate` to playback the recorded events. We next define few terms before explaining the architecture of recording and playback mechanism.

*Event* is an occurrence of an interaction between the user and the windowing system. The windowing system constitutes of the local display, the keyboard, and the mouse. *Recording and playback* essentially involves capturing all events that are generated due

to user actions during a session, and reproducing those events in exactly the same sequence as they were generated.

(a) Typical timing diagram of event generation



(b) Events and its timing information stored

Event list	Event details	Start times
$E_0$	...	$t_{r_0}$
$E_1$	...	$t_{r_1}$
$\vdots$		$\vdots$
$E_n$	...	$t_{r_n}$

Fig. 4. Details of event generation and timing.

Figure 4(a) shows generation of various events and its associated timing information. The X-axis depicts time increasing from left to right, whereas the length of each event shown represents the amount of time it takes to process an event after it has been generated. The terminology used in the illustrated timing diagram is as follows:

- $E_i$  The  $i^{th}$  event in a session.
- $t_{r_i}$  The instant at which the event  $E_i$  occurs.
- $t_{r_{i+1}} - t_{r_i}$  The time difference between the occurrence of the event  $E_{i+1}$  and the event  $E_i$ .
- $n$  The total number of events generated during a session.

We classify events generated into two categories: (1) *window events* are those generated as a direct result of the user’s interaction with the application, and (2) *synthesized events* are those which emulate the window events using the Tcl/Tk commands from within the application program and are *not* a result of user actions.

Every window event consists of at least one primitive component. Examples of primitive components include: ButtonPress, ButtonRelease, MouseMotion,

KeyPress, etc., and are used to identify the type of event that is occurring. However, additional information may be necessary to fully describe an event and is available as a secondary component. Secondary components represent details such as the x-y coordinates of the mouse cursor on the screen, the name of key that was pressed, the button number of the mouse that was clicked, etc. The primary and secondary components can then be used together to generate a synthesized event from within the Tcl application. Thus, Figure 4(b) shows a list of events generated and its primary and secondary components along with the timing information that is necessary to synthesize a window event.

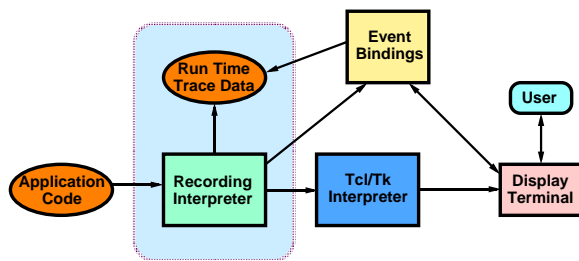


Fig. 5. Architecture of recording of a session.

**Recording Session Architecture.** Figure 5 shows the architecture of how to record an entire session consisting of all the user interactions with the Tcl application. It is necessary to intercept and capture the entire sequence of user actions and save it in a file as a *run time trace data*. This involves two steps:

1. A *recording interpreter* intercepts every Tcl/Tk command of the application that is being passed to the Tcl interpreter and dynamically associates a new class of binding called *RecordTaker* to every widget that is being created for the application. Thus, for example, if a button widget called ‘.b’ is being created by the application, then we use the following command to associate the *RecordTaker* bind class with it:

```
bindtags .b [linsert [bindtags .b] 0 RecordTaker]
```

2. Whenever a user interacts with the Tcl application, the binding script for the *RecordTaker* is invoked first, before processing any of its default bindings. The binding script for *RecordTaker* is designed to capture all the information relevant to a specific widget and save it as the *run time trace data*. A typical run time trace data that is saved in a file is shown in Figure 4(b).

**Playback Session Architecture.** Figure 6 shows the architecture of playback of a recorded session which is stored in a file containing the *run time trace data*. The playback of a recorded session is

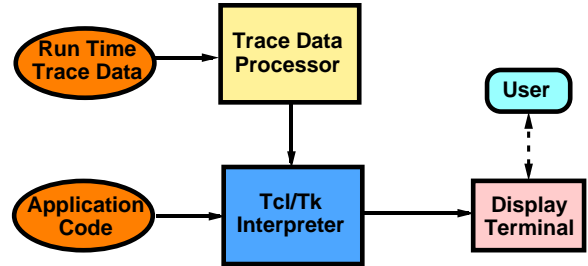


Fig. 6. Architecture of playback of a session.

initialized by invoking the original Tcl/Tk application code with the Tcl interpreter. The *trace data processor* then reads the *run time trace data* from the specified file and processes it to create synthesized events. The timing information associated with each event recorded is very important and critical in synchronization of the synthesized event during playback. It is possible to playback an exact replica of the recorded session, provided the cpu load is not significantly different from that during the recording process. The effect of variations in cpu load is as follows:

1. If the cpu load has increased during playback, then it will simply result in longer time for completion of the playback session.
2. On the other hand, if the cpu load has significantly decreased during playback, then some of the synthesized events may be generated even before processing of few of the earlier events has completed. This can result in generation of synthesized events which are not in original sequence and hence the playback session may fail.

Based on these experiences, we decided to allow the user to control the speed of execution during playback. We next describe the mechanism for scheduling synthesized events that facilitates the user to achieve varying speed of playback session.

**Event Scheduling.** We first define the terminologies used for scheduling the events of a playback session:

- $t_{p_i}$  The time at which the synthesized event  $E_i$  is scheduled for playback.
- $s$  The scaling factor which determines the speed of the entire playback session. It is constant for all the  $n$  events and is pre-computed before the commencement of a playback session.
- $s_i$  This is a dynamic scaling factor for the  $i^{th}$  synthesized event. The value of this scaling factor is controlled by the user and can change during the playback session.

Figure 7(a) and (b) show two schemes of scheduling synthesized events. Both the schemes use the Tcl command `after` to generate event at a specified

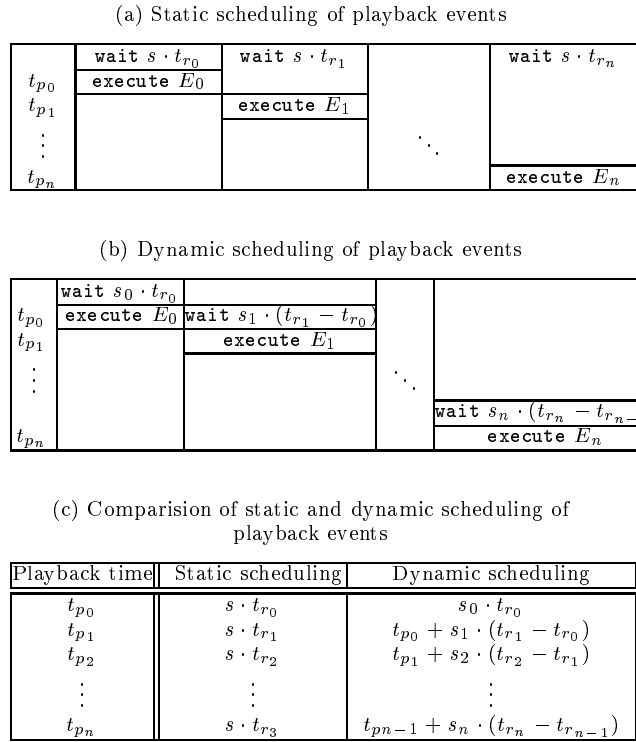


Fig. 7. Scheduling playback of recorded events.

instant. In the static scheme, all the  $n$  events are scheduled for playback as soon as the Tcl application is initialized. Therefore, the speed of the playback session has to be decided in advance and cannot be changed later on during actual playback. On the other hand, in the dynamic scheme, each event is scheduled for playback as soon as the idle time interval of its previous event is completed, as shown in Figure 7(b). At the end of time interval  $s_0 \cdot t_{r_0}$ , it not only generates the event  $E_0$ , but also schedules the `wait  $s_1 \cdot (t_{r_1} - t_{r_0})$`  interval for generating the next event  $E_1$ . This feature gives the user the flexibility to dynamically control the speed of playback execution by changing the value of the scale factor  $s_i$ . In addition, it is also possible to pause the execution of the playback session by merely deferring the scheduling of the next event.

**Implementation Example.** We use a simple application `Print Hello` button in Figure 8 to illustrate the main ideas used to implement the recording and playback mechanism.

The left side of the figure shows the trace data, and the right side of the figure shows the Tcl/Tk commands used for synthesis of the recorded events and the user views as each event is synthesized. The following steps are necessary to invoke the command associated with the button widget:

Trace Data From  
a Recording Session

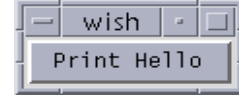
Application code

Window : .b  
Event : Enter  
Time :  $t_{r_0}$

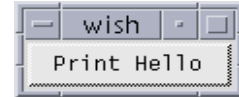
Window : .b  
Event : Button-  
Press  
Time :  $t_{r_1}$   
Mouse button : 1

Event Synthesis  
during Playback Session

```
pack [button .b -text
"Print Hello" -command
"puts Hello"]
```



```
event generate .b
<Enter>
```



```
event generate .b
<ButtonPress> -button
1
```

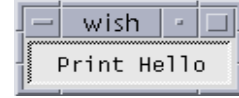


Fig. 8. Details of a recording and playback session.

*Step 1.* Initialize the application code for displaying the button widget with the command

```
pack [button .b -text "Print Hello" \
      -command "puts Hello"]
```

*Step 2.* It is necessary to activate the button widget before invoking its command. Therefore, synthesize the event 'Enter' in the window '.b' with the command `event generate .b <Enter>`.

*Step 3.* Next, synthesize the event 'ButtonPress' in the window '.b' with the command `event generate .b <ButtonPress> -button 1`. This results in printing of the text "Hello" to the standard output.

**Recording and Playback Tools.** Figure 9 shows the GUI of two tools - a *RecordTaker* that assists users to create customized recordings, and a *PlaybackMaker* for playback of a recorded session with capabilities to control its speed or pause as desired. The *RecordTaker* also allow the users to record a session as a series of several smaller steps, called *frames*, instead of one single large recording. In addition, a user can type in appropriate text for each frame describing its functionality.

Thus, once a session is recorded in several frames, the *PlaybackMaker* automatically stops at the completion of each frame, displays the associated text for

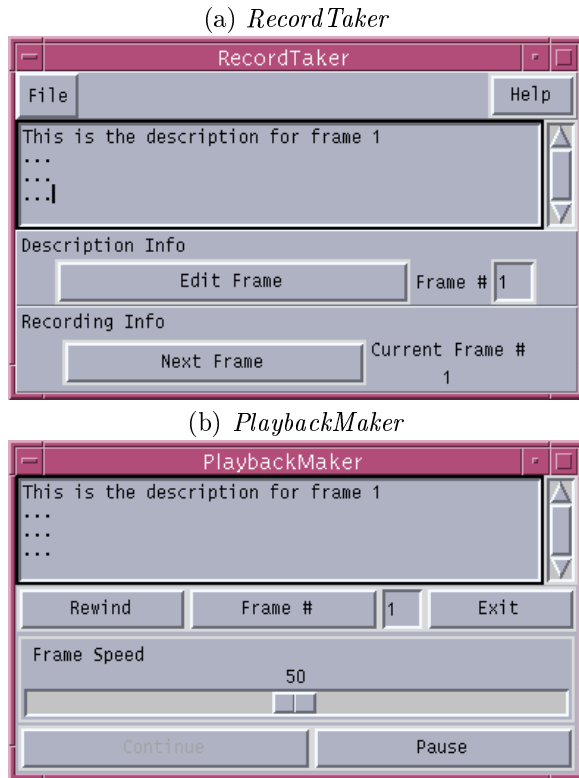


Fig. 9. Recording and playback tools.

the user to read and waits for the user to click on the ‘Continue’ button to start processing the next frame. This feature is very useful for recording tutorial sessions, where each tutorial consists of several frames with a text describing each frame during playback.

## 5 Experiments

The prototype of an environment that records, plays back and executes a Tcl/Tk collaborative Internet-based desktop, is being put to the test as an integral part of a national-level collaborative and distributed design project involving teams at 6 sites [12]. Specifically, the desktop brings together distributed data, application workflows, and teams into collaborative sessions that share the control of the desktop editing and execution. A typical workflow, such as the one shown in Figure 10, invokes distributed tools and data to support a major phase in the design of microelectronic systems. A detailed description is available in [8, 10].

We argue that recording and playback of collaborative user interactions can have a wide-range of applications, such as: ‘keeping minutes’ of interactive discussions, clicks of menu-specific commands associated with different tools on the shared desktop, user-entered data and control inputs, user-queried

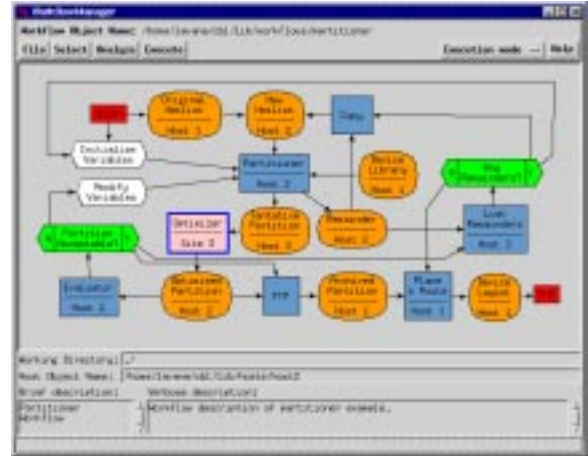


Fig. 10. Partitioner workflow.

data outputs, support for automated software documentation, tutorials, collaborative playback of tutorials and solutions recorded earlier, etc. The 540 experiments, summarized in this section, are the initial part of the Internet desktop environment performance and functionality evaluation, conducted before its release to Vela Project participants and others.

Each of these experiments relies on *interactive* user inputs. To maintain consistency of user inputs during the repeated trial executions across the Internet (with variable quality-of-service), we first *record* a single reference instance of each test case on the local server (without relying on the network) and then move these recordings to cross-state and cross-country servers on the Internet. Each server has an executable version of *ReubenDesktop*, *OmniBrowser*, *RecordTaker*, and *PlaybackMaker*. The experiments are initiated with a *playback* that executes recorded instances of test cases, multi-casting them to 1, 2, or 3 workstation displays at CBL. Additional details about these tools are available in [8, 9, 11]. Experiments reported in this section support a conjecture that will be the subject of more detailed experimentation later:

*Task-specific performance of a single/multiple client-server ReubenDesktop execution can be predicted, under comparable server and network loading, by measuring the performance of pre-recorded task-specific experiments that are executed and multi-cast by the server to one/multiple client displays.*

In other words, to assess the performance of *interactive distributed sessions that involve one or more participants*, we have verified that the experiments, as reported in this section, can be extrapolated by measuring the performance of single- and multi-

cast executions that are based on playback of pre-recorded experiments on a reference server. The benefits of not requiring a number of individuals to sit through repeated session experiments are obvious. Specifics about the testbed configurations, test cases considered, and graphical results follow.

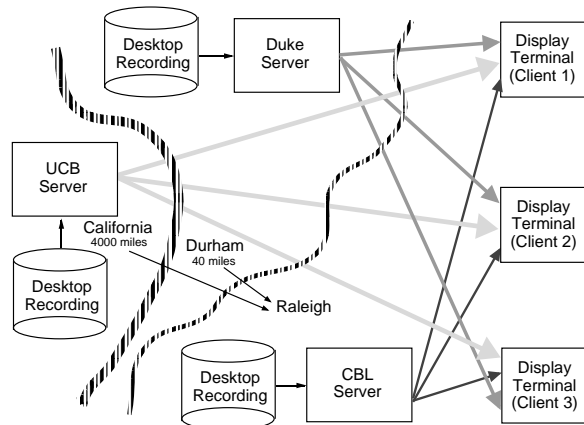


Fig. 11. Test-bed configuration for experiments.

**Testbed Configurations.** In order to approximate typical instances of a distributed multi-site collaborative desktop environment, we used a test-bed setup, as shown in Figure 11, to create:

- (1) a *local environment* by installing the desktop software on a CBL server<sup>1</sup> at NC State University in Raleigh, NC, which is multi-casting its desktop to one or more CBL client hosts;
- (2) a *cross-state environment* by installing the desktop software on a server<sup>2</sup> at Duke University in Durham, NC, which is multi-casting its desktop to one or more CBL client hosts; and
- (3) a *cross-country environment* by installing the desktop software on a server<sup>3</sup> at the University of California in Berkeley, CA, which is multi-casting its desktop to one or more CBL client hosts.

We have carefully selected two remote servers such that the physical distance of approximately 40 miles and 4000 miles from Raleigh to Durham and Berkeley respectively, represents a realistic test-bed for performance evaluation. The arrows, shown in Figure 11, depict broadcasting of the desktop environment from each server to the three display terminals at CBL in Raleigh.

**Test Cases.** We have created and *recorded*, directly on the CBL server under negligible loading conditions, six test cases of collaborative sessions with useful attributes that demonstrate typical user-

invoked tasks. The brief description that follows includes the reports of *real\_time*, *user\_time* and *system\_time* as produced by the Unix utility *time*. The ‘real\_time’ corresponds to the ‘stopwatch\_time’ that could have been obtained by the user monitoring the task. The ‘user\_time’ is the time required by the CPU to complete the task. The ‘system\_time’ is the CPU time required by the system on behalf of the task. A brief description of all test cases engaging two participants, that were recorded for the experiment, follows.

(1) *Co-editing-1* (*real\_time*=119.4s, *user\_time*=31.1s, *system\_time*=1.5s): Using *ReubenDesktop*, we *open, and edit*, a simple 4-node, 3-arc workflow by selecting, opening, and closing a single data file node-configuration window.

(2) *Co-editing-2* (*real\_time*=153.1s, *user\_time*=44.0s, *system\_time*=1.9s): Using *ReubenDesktop*, we *open, and edit*, the same 4-node, 3-arc workflow by selecting, opening, and closing a single data file node-configuration window and a single program node-configuration window.

(3) *Co-editing-3* (*real\_time*=223.8s, *user\_time*=67.5s, *system\_time*=2.5s): Using *ReubenDesktop*, we *open, and edit*, the 17 node, 22 arc workflow by selecting, opening, and closing 3 data files and a single program node-configuration window.

(4) *Co-browsing-1* (*real\_time*=136.7s, *user\_time*=56.1s, *system\_time*=2.1s): Using *OmniBrowser*, we *traverse* a directory structure, located on the server’s local file system, across 3-levels, with up to 141 items in each directory. The directory structures of all the three servers were made exactly the same for uniform comparison.

(5) *Co-browsing-2* (*real\_time*=159.2s, *user\_time*=97.5s, *system\_time*=5.0s): Using *OmniBrowser*, we *select, open, and scroll*, from start to end, the same copy of a text file of about 1000 pages (2.2Mb), located on each server.

(6) *Co-execution-1* (*real\_time*=123.9s, *user\_time*=90.0s, *system\_time*=3.8s): Using *ReubenDesktop*, we *open, and execute*, the hierarchical workflow in Figure 10. As shown, the workflow has 22 nodes and 28 arcs; during execution, the node labeled as *optimizer* expands into a sub-workflow with 14 nodes and 15 arcs. All test cases involved two participants working collaboratively and consisted of exchanges of several dialogs via the *FlowSynchronizer* between the two, during each recording session.

**Evaluation Method.** All software and the files of six test cases, recorded directly on the CBL server, have been replicated on the server at Duke U. and the server at UCB. Scripts have been invoked, *during the night when both servers and the network were*

<sup>1</sup>SUN SPARC 20 (chip=60MHz memory=64Mb swap=732Mb)

<sup>2</sup>SUN Ultra 1 (chip=167MHz memory=256Mb swap=288Mb)

<sup>3</sup>SUN SPARC 20 (chip=60MHz memory=96Mb swap=365Mb)

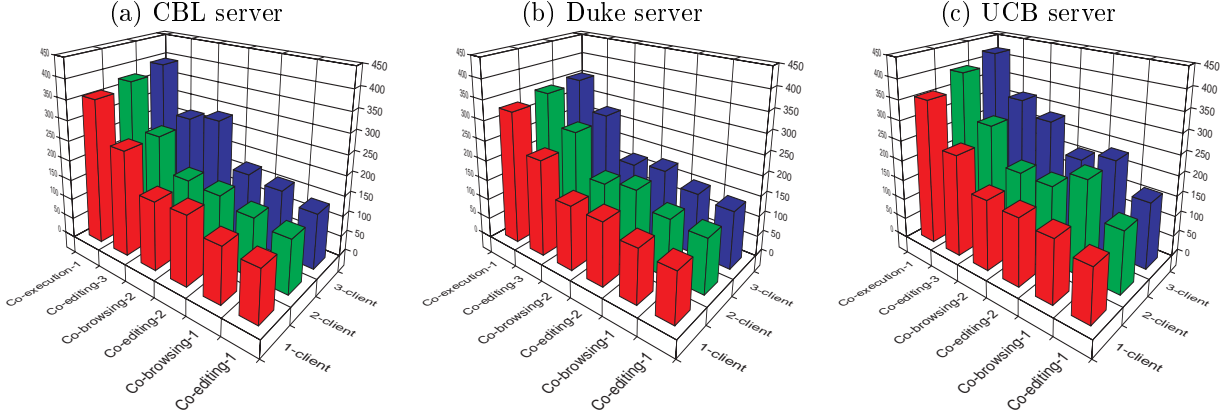


Fig. 12. Evaluation results of 540 experiments on three servers.

least loaded, to execute the 540 experiments as follows:

From each of the three servers, execute and multicast 10-times, with interval of 30 seconds between each execution:

(1) successively to one, two, and three client hosts at CBL, recordings of *co-editing-1*, *co-editing-2*, *co-editing-3*;

(2) successively to one, two, and three client hosts at CBL, recordings of *co-browsing-1*, *co-browsing-2*;

(3) successively to one, two, and three client hosts at CBL, recording of *co-execution-1*.

A log file, generated by `time` (`real_time`, `user_time`, `system_time`) command, archives timing data for each experiment. Similarly, a log file, generated by `sar` (`system activity report`) command, archives the load on each of the three servers during the execution of these experiments. The log file generated by `sar` provided the information whether or not both the load on the server and the network was sufficiently stable to accept the ‘`real_time`’ and ‘`user_time`’ results for tabulation.

The data generated as a result of evaluating the six test cases are tabulated and plotted as a 3-dimensional graphs shown in Figure 12 for each server. The x-axis in each plot lists the name of the test case and the y-axis represents the number of clients that a specific test case was multi-casted to. The time required for execution of each test case with 1-, 2- and 3-clients is then represented as a bar on the z-axis.

**Summary of Results.** The data plotted in Figure 12 allows us to evaluate the performance of Internet-based desktop environments.

1. The ‘`real_time`’ for playback to a single-client on the reference server is approximately the same as the time required to record the test cases.

2. The ‘`real_time`’ for playback from other servers varies, depending on the distance between the host

server and its clients and the characteristics of the host server. Specifically, for single-client playback, Duke server consistently reported least execution times, followed by CBL server and UCB server. This is attributed to the higher performance server at Duke. However, for multi-clients, the execution times increased with distance in the order CBL, Duke, and UCB.

3. When the experiment is multi-cast to 2-clients or 3-clients, it takes slightly more time, on the order of a few seconds, for execution than the time required for single client execution. The negligible increase in the playback time for multi-client execution is due to the fact that the exchange of dialog among participants is computationally least intensive.

4. The variations in minimum and maximum values of ‘`real_time`’ for each experiment are negligible since the experiments were performed during the night. However, the same experiments showed significant variations during the day when the network traffic and the server load are unpredictable.

5. Comparing the ‘`user_time`’ and the ‘`system_time`’ for each server, we find that the CBL server requires the most CPU time and the Duke server requires the least CPU time. This follows directly from the different types of processors and the configuration of each server.

**Observations.** The successful completion of all 540 experiments provides us with assurance that the experiments are consistently reproducible on a variety of servers, given that the server nominal load is small and that the network is stable. Specifically, we confirmed that

- Repeated *real time* executions of experiments, where user-inputs are carefully and consistently entered (rather than pre-recorded), gives ‘`real_time`’, ‘`user_time`’, and ‘`system_time`’ performance that is comparable (within 10%) to the times reported for

pre-recorded execution on any server – provided that the server load and network conditions are as favorable.

- The performance of the Internet-based desktop environment, even in a *collaborative mode*, is quite good under nominal network traffic and load on the server. Hence, with sufficient network bandwidth and powerful processors, it is possible to work collaboratively with efficiency and effectiveness even when participants are dispersed across the continent.

- As the number of clients, corresponding to each participant, increase from 1 to  $n$  during playback, the increase in ‘real-time’ execution is on the order of a few seconds only. Again, this increase is subject to the server and network performance and the amount of dialog among participants present in the recording.

## 6 Software Availability and Status

The collaboration, recording and playback features are all currently integrated into a single desktop environment. We are planning to unbundle the desktop environment and make them available as separate packages:

(1) *CollabTclTk*. Collaboration of *any* tcl-application to multiple sites,

(2) *RecordnPlayTclTk*. Recording and playback session of *any* tcl-application, and

(3) *ReubenDesktop*. Internet-based desktops integrated with collaboration and recording/playback packages.

For more details about the current status of these packages, please visit:

<http://www.cbl.ncsu.edu/software>

## 7 Conclusions and Future Work

We have introduced a new paradigm and a prototype implementation of a collaborative and recordable environment on the Internet using Tcl/Tk. Complementing the objectives of the user-reconfigurable Internet-based desktop environment, this environment supports

- *peer-to-peer* interaction between members of any team;
- *peer-to-workflow* interaction between any team member to any object in the workflow;
- *recording and playback* of interactive execution of Tcl/Tk applications and collaborative sessions.

Future work on collaborative and recordable environments should address the following issues:

*Security of collaboration*. Our current implementation is ‘insecure’ since it relies on participating hosts to open their X displays for remote connec-

tion. Users who are behind company ‘firewalls’ are not able to participate, given the present set-up.

*Dynamic collaborative team*. Currently, we have to close and restart the application for changing the number of participants. In the future, we want to support dynamic addition and removal of users participating in collaboration.

*Editing of the recording session*. Recording, *without errors*, of a long complex session is difficult. Hence, recording sessions of shorter durations and slicing them together later for playback would be more effective.

ACKNOWLEDGMENTS. We could not have reported as comprehensively on the results of our Internet Desktop experiments without getting generous user accounts on two remote servers, facilitated by Dr. Richard Newton at UC Berkeley and Dr. Gershon Kedem at Duke U. We thank them and their support staff for this privilege.

## References

- [1] Scriptics Corporation. Published under URL <http://www.scriptics.com/>, 1998.
- [2] The Tcl/Tk Consortium. Published under URL <http://www.tclconsortium.org/>, 1998.
- [3] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
- [4] M. Rosenman and S. Greenberg. Designing Real-Time Groupware with GroupKit, A Groupware Toolkit. In *ACM Transactions on Computer Human Interaction*, 1996.
- [5] GroupKit. Published under URL <http://www.cpsc.ucalgary.ca/grouplab/groupkit>, 1997.
- [6] Charles Crowley. TkReplay: Record and Replay in Tk. In *Proceedings of the Tcl/Tk Workshop, Toronto, Canada*, July 1995.
- [7] D. Libes. *Exploring Expect*. O’Reilly and Associates, 1995.
- [8] H. Lavana, A. Khetawat, F. Brglez, and K. Kozminski. Executable Workflows: A Paradigm for Collaborative Design on the Internet. In *Proceedings of the 34th Design Automation Conference*, pages 553–558, June 1997. Also available at <http://www.cbl.ncsu.edu/publications/#1997-DAC-Lavana>.
- [9] Amit Khetawat. Collaborative Computing on the Internet. Master’s thesis, Electrical and Computer Engineering, North Carolina State University, Raleigh, N.C., May 1997. Also available at <http://www.cbl.ncsu.edu/publications/#1997-Thesis-MS-Khetawat>.
- [10] H. Lavana, A. Khetawat, and F. Brglez. Internet-based Workflows: A Paradigm for Dynamically Reconfigurable Desktop Environments. In *ACM Proceedings of the International Conference on Supporting Group Work*, Nov 1997. Also available at <http://www.cbl.ncsu.edu/publications/#1997-GROUP-Lavana>.
- [11] H. Lavana, A. Khetawat, and F. Brglez. REUBEN 1.0 User’s Guide. CBL, Research IV, NCSU Centennial Campus, Box 7550, Raleigh, NC 27695, 1998. To be available at <http://www.cbl.ncsu.edu/publications/>.
- [12] Globally distributed microsystem design: Proof-of-concept. A university-based project involving teams at 6 sites. See the project home page at <http://www.cbl.ncsu.edu/vela> for more details.