

CollabWiseTk:

A Toolkit for Rendering Stand-alone Applications Collaborative

Hemang Lavana Franc Brglez

CBL (Collaborative Benchmarking Lab), Dept. of Computer Science, Box 8206

NC State University, Raleigh, NC 27695, USA

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

Abstract

Traditionally, a stand-alone client application is rendered collaborative for members of a team either by sharing its view or by re-writing it as a collaborative client. However, it may not be possible to anticipate in advance all preferences for collaboration, hence such a client may appear confusing to some of the team members.

We propose a novel client/server architecture for tk-based applications: rendering any stand-alone client collaborative, without a code re-write. Participants themselves are allowed to dynamically re-configure the inter-client synchronization table to suit their changing preferences and needs. The CollabWiseTk toolkit, based on the proposed architecture, is an extension of the tk functionality to support collaboration. It re-defines the existing tk commands such that the entire tk widget set is rendered collaborative for use with multiple users.

We demonstrate the capabilities of the CollabWiseTk toolkit by readily rendering collaborative most of the Tk Widget Demonstrations, distributed with the core Tcl/Tk. The toolkit is implemented in pure tcl and it ports to all platforms.

Keywords: Internet, Collaboration, Groupware, Tcl/Tk, GUI.

1 Introduction

This paper is one of the two companion papers [1] that were initiated at the conclusion of the course on *Frontiers of Collaborative Computing on the Internet* (csc591-b, [2]).

A number of collaborative client/server architectures have been proposed to date. Principally, they deal with specific applications ranging from a shared calendar (The Electric Secretary) [3] to a shared whiteboard [4]; from collaborative visualization for health

care [5] to collaborative editing of schematic diagrams [6]. An architecture that supports workflows of heterogeneous applications is described in [7, 8, 9]. Some architectures expect that the application has been written for a team of users, e.g. the GroupKit architecture [4]. Alternatively, multi-casting can render an application written for a single user collaborative, e.g. the REUBEN architecture [7, 8, 9]. There are disadvantages to both approaches: the need to write an application for multiple users, and the performance issues of multi-casting.

Most of the client applications today are as stand-alone applications. The traditional approach is to re-write it as a client for collaborative application. This can be a formidable task, especially when all possible preferences for modes of collaboration cannot be anticipated in advance. Such a client may turn out to be user-unfriendly or confusing for a particular team. Simple preferences, such as whether and when should the scrollbars track for all participating collaborators, or should separate scrollbars be provided (and color-coded) for each participant, are at the core of such issues [4, 10, 11].

In this paper, we propose a novel client/server architecture for tk-based applications: rendering any stand-alone client collaborative, without a code re-write. Participants themselves are allowed to dynamically re-configure the inter-client synchronization table to suit their changing preferences and needs. The CollabWiseTk toolkit, based on the proposed architecture, is an extension of the tk functionality to support collaboration.

The paper is organized into following sections:

- Background and Motivation;
- CollabWiseTk Architecture;
- Inter-client Synchronization;
- CollabWiseTk Implementation;
- Testbed and Experiments;
- Software Evaluation;
- Software Availability and Status;
- Conclusions.

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).

2 Background and Motivation

Let us consider a very simple application which consists of a text widget with a vertical scrollbar. Such an application allows the user to type in text and the vertical scrollbar allows the user to browse the text information, when the size of the text widget is not large enough to display the entire text at the same time. This application can be very easily built using four lines of tcl code, as shown in Figure 1(a), and is a basic widget used by many complex applications that need functionalities such as syntax highlighted message display, text editing, and html display, to name a few.

Several possibilities exist even for a simple text widget that is rendered collaborative. Figures 1(b) - (e) shows various possible collaborative configurations of a text widget for two users Alice and Bob.

Figure 1(b) shows Alice and Bob sharing the same view of the text widget as well as the scrollbar. A centralized server ensures that both the views are synchronized at all the times. Possibilities of conflict arise when Alice and Bob both try to interact with the text widget at the same time. Such conflicts are typically resolved by some form of locking mechanisms, such as round-robin, first-come-first-serve, user-controlled token passing, etc, so that only one person can interact with the application at a time while the others are forced to watch.

Figure 1(c) shows a distributed implementation of a collaborative text widget that allows Alice and Bob to interact with their individual text widgets, while an event synchronizer dispatches these interactions to the other users. This mechanism also allows participating users to have different views of the same widget and occasionally ‘glance’ at the other widgets.

Figure 1(d) shows a configuration where both Alice and Bob get the same view of the text widget, but each has a personal edit cursor so that both can type in simultaneously without affecting the other. In the example shown, Bob has an edit cursor at the top, while Alice has an edit cursor at the bottom and hence both can work on different sections of the same text file. However, the associated scrollbar needs to be configured such that when Bob changes the scroll-view, Alice may not want to follow Bob’s scrolled movement when she’s editing, but may want to do so when merely watching Bob’s interactions.

Figure 1(e) shows yet another configuration

where the text widget is replicated once for every participating user. Therefore, Alice and Bob both have two text widgets - one where each can type in text and another where each can observe what the other is typing. Here again, Bob may prefer the scrollbars to be synchronized with Alice, while Alice may want to scroll independently Bob’s text widget. This is equivalent to widely available chat tools or the Unix ‘talk’ utility.

All of the above examples can be easily implemented in tcl by re-writing the four lines of stand-alone tcl code. However, when this text widget is part of a more complex application containing several other widgets, each of which can themselves have their own numerous configuration possibilities, it becomes very difficult to anticipate in advance a suitable configuration preference. Such a collaborative application might turn out to be user-unfriendly or confusing for a particular team.

3 CollabWiseTk Architecture

The CollabWiseTk toolkit consists of two parts:

A synchronizing group server (SGS) that provides mechanisms for communication and synchronization among multiple-user client applications; and

A distributed collaboration client that provides mechanisms for inter-client synchronization among multiple-user client applications for effective collaboration.

The general architecture of the toolkit is shown in Figure 2. This architecture extends and complements the *Asynchronous Group Server Architecture* (AGS) in [1].

SGS is a tcl server that accepts socket connections from various collaboration clients. It has three types of repositories:

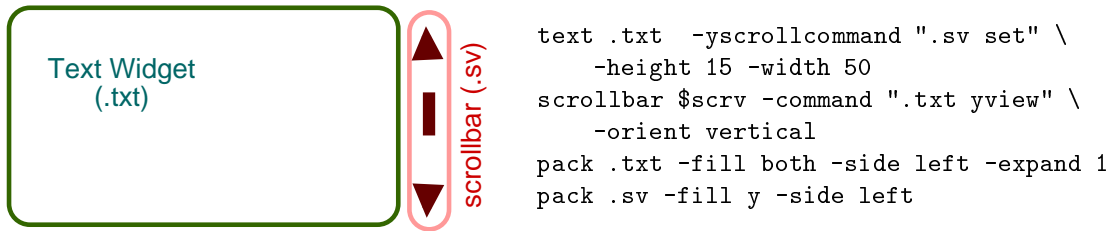
Tcl scripts and packages: various stand-alone tcl applications are deposited here and are available to users for collaboration;

Inter-client synchronization tables: different configuration preferences for a tcl application are stored in this tables; and

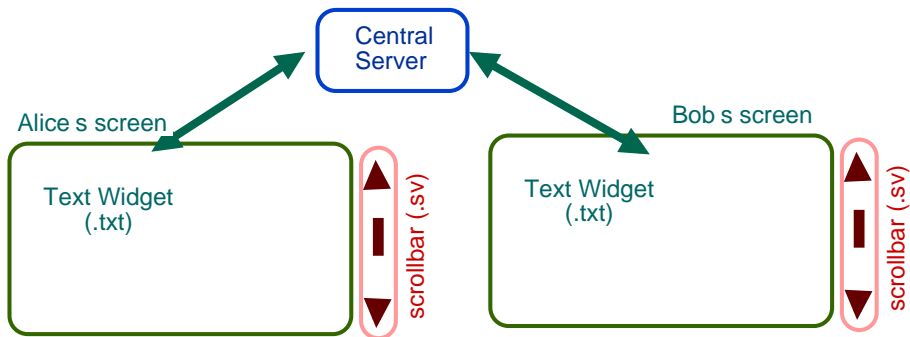
Registered users and access permissions: for security reasons, the latter maintains a list of registered users and their corresponding access privileges.

The collaboration client is installed on each user’s machine and provides an interface to the user for collaborating with other users. Upon invocation, the collaboration client establishes a socket connection

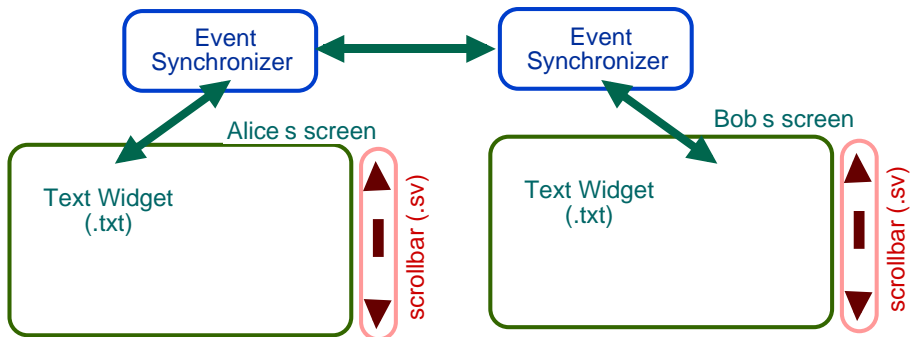
(a) Simple text widget with a vertical scrollbar and its tcl code.



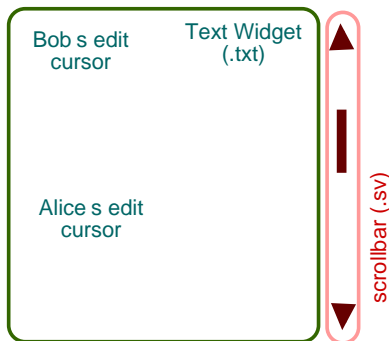
(b) Synchronized collaborative view using centralized server.



(c) Collaborative view using distributed architecture.



(d) Collaborative shared editor (for source code).



(e) Collaborative chat box (like talk window).

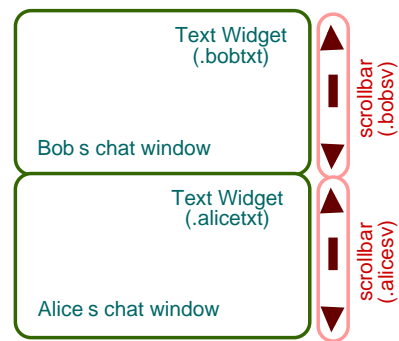


Fig. 1. Desirable collaborative configurations for a text widget with a scrollbar.

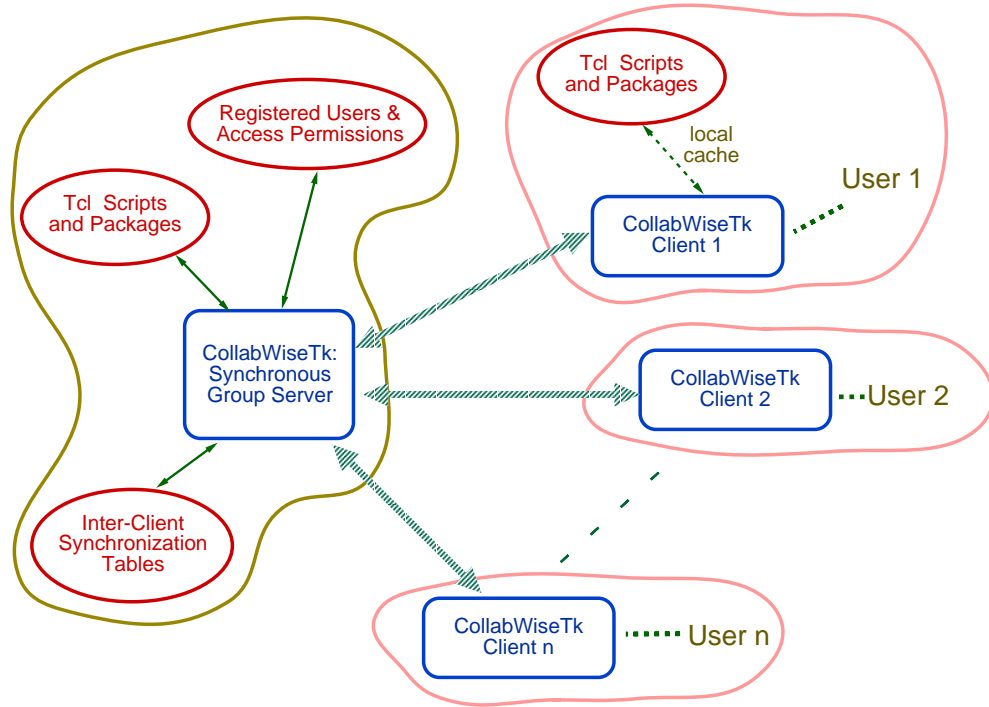


Fig. 2. A high level view of client/server architecture for collaboration.

with SGS and prompts the user to identify herself. Once the login process is completed, the user can access several different tcl applications, based on her access privileges, by invoking an appropriate configuration file from the inter-client synchronization table. Collaboration clients also maintain a local cache of the tcl applications for faster access.

The collaboration client also provides mechanisms to install new tcl scripts and packages in the group server repository. Privileged users can install such scripts and easily create configuration files on the server for others to access.

When two or more clients access the same configuration file of a tcl package, they are immediately set-up for collaboration. User-interactions with the tcl application are then sent to the group server, which in turn relays this information to all participating users.

4 Inter-Client Synchronization

Inter-client synchronizer allows the user to dynamically re-configure the different modes of collaboration for every primitive object contained within the tcl application GUI. A primitive object is an element of GUI with which a user can either interact

or observe: (1) all tk widgets, such as button, label, entry, etc, are primitive objects; and (2) in addition, tagged items of a canvas and text are also considered as primitive objects.

Primitive objects can be configured into either one of the two states - *interact* or *observe*. When a primary object is configured to be in *observe* state, a user is prevented from interacting with it. For example, a user can be prevented from interacting with a text widget by removing all of its binding tags, as follows:

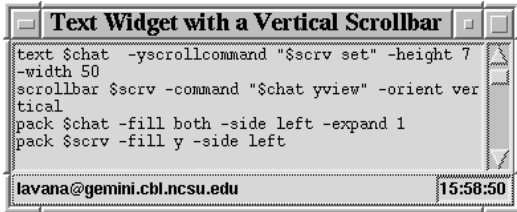
```
bindtags .txt {""}
```

The same text widget can be brought back to an *interact* state merely by restoring its binding tags. A good introduction to *bindtags* can be found in [3].

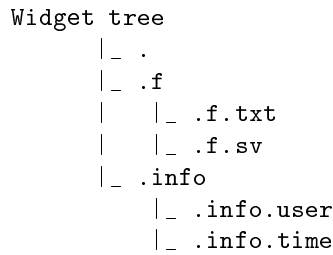
When several users work collaboratively, each user invokes the tcl application locally. Therefore, all primitive objects of an application are replicated on each user machine. In addition, a user can now configure her primitive object to be in one of the two states, *interact* and *observe*, and link it to any of the participating users. This means that when there are two users, say Alice and Bob, each can configure their text widget to be in one of the four states:

- Interact/Alice,
- Observe/Alice,
- Interact/Bob, and
- Observe/Bob.

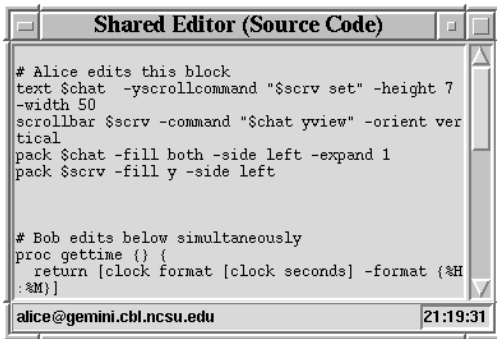
(a) Simple text widget with a vertical scrollbar.



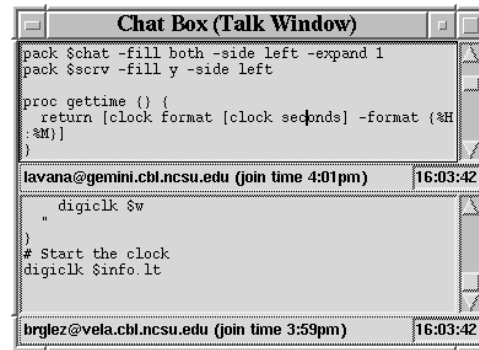
(b) Widget tree structure



(c) Collaborative shared editor (for source code).



(d) Collaborative chat box (like talk window).



(e) Inter-client synchronization table

Network / Login	Inter-Client Synchronizer	Install packages, configure users, build p
<ul style="list-style-type: none"> Widget Demos <ul style="list-style-type: none"> Text Widget Shared Editor <ul style="list-style-type: none"> .f <ul style="list-style-type: none"> .txt .sv .info Scrollable Text Box <ul style="list-style-type: none"> .f <ul style="list-style-type: none"> .txt .sv .info Chat Box <ul style="list-style-type: none"> .bobf <ul style="list-style-type: none"> .txt .sv .bobinfo .alicef <ul style="list-style-type: none"> .txt .sv .aliceinfo Hello World Chat Rooms 	alice@host.domain.com Interact / Alice Interact / Alice Interact / Alice Interact / Alice Interact / Alice Interact / Alice alice@host.domain.com Interact / Alice Interact / Alice Interact / Alice Interact / Alice Interact / Alice Interact / Alice alice@host.domain.com Interact / Alice Interact / Alice Interact / Alice Interact / Alice Interact / Alice Interact / Alice	bob@host.domain.com Interact / Bob Interact / Bob Interact / Alice Interact / Bob Interact / Bob Interact / Bob bob@host.domain.com Interact / Bob Interact / Bob Interact / Bob Interact / Bob Interact / Bob Interact / Bob bob@host.domain.com Interact / Bob Interact / Bob Interact / Bob Interact / Bob Interact / Bob Interact / Bob

Standard Output and Error

Fig. 3. Inter-client synchronization table used to configure a text widget into various collaborative modes.

We next explain the meaning of each of this state for Alice:

Interact/Alice: Alice is configured to interact with her own text widget. At the same time, Bob can observe her interactions, if he has configured his text widget to be in *Observe/Alice* state.

Observe/Alice: Alice is configured to observe her own text widget. This would result in activities on the text widget, unless Bob has configured himself to be in *Interact/Alice* state.

Interact/Bob: Alice is configured to interact with Bob’s text widget. However, if Bob is also configured to be *Interact/Bob* state, then this could lead to potential conflicts.

Observe/Bob: Alice is configured to observe Bob’s text widget.

The simple text widget example with a vertical scrollbar, shown in Figure 1(a), can be easily configured into various states by defining appropriate object state for the two widgets. Figure 3(a) shows a snapshot of a text widget GUI and Figure 3(b) shows its corresponding widget tree. This application is readily transformed, without any re-write, into: (1) collaborative shared editor for source codes (Figure 3c), and (2) a chat box window (Figure 3d), by providing an appropriate configuration file.

Figure 3(e) shows a snapshot of the inter-client synchronization table. It lists all the configuration files for the text widget in the left column, with their respective widget trees. Collaborative participants, if any, are listed in the adjacent columns. Thus, there are two participants for ‘Shared Editor’ and ‘Chat Box’, and only one participant for ‘Scrollable Text Box’. The entries listed below each user corresponds to the current state of the respective widget. For example, the text widget ‘.txt’ under ‘Shared Editor’ is listed as *Interact/Alice* for both the users, ‘alice@host.domain.com’ as well as ‘bob@host.domain.com’. This implies that while Alice is interacting with her text widget, Bob also has the permission to simultaneously interact with Alice’s text widget. The text widget is therefore shared among the two users thereby providing a means of real-time collaboration. Each such entry can be changed to a different state merely by clicking on the dropdown menus and selecting the desired state. We have used BWidget toolkit [12] to implement the GUI shown in Figure 3(e).

The examples shown above describe how each widget can be configured to suit one’s requirements. How-

ever, as the size of the application grows, it can become very tedious for a user to configure each of this widget individually. Therefore we also provide a mechanism whereby if a user configures one widget to a specific state, then all of its subsequent children widgets also assume the same state. This becomes very useful when the user wants to start or stop interacting with the entire toplevel window and can be achieved by merely changing the toplevel window to the appropriate state.

5 CollabWiseTk Implementation

The client-server architecture of the CollabWiseTk toolkit is implemented using socket programming. Typically several clients may be connected to SGS at any time. In addition, clients may invoke several tcl applications and each tcl application may have several different collaborative participants. Figure 4 depicts one such scenario, where user1 on client1 has invoked TkAppln1, TkAppln3 and TkAppln4, user2 on client2 has invoked TkAppln2 and TkAppln4 and user3 on client3 has invoked TkAppln1 and TkAppln2. An application that is common among users imply that those users are its collaborative participants. For example, TkAppln1 is rendered collaborative among user1 and user3. This is also shown as a dashed line linking TkAppln1 to client1 and client3 on the server side.

The following table provides such a relationship for the example shown in Figure 4:

Application/Username	User1	User2	User3
TkAppln1	*		*
TkAppln2		*	*
TkAppln3	*		
TkAppln4	*	*	

As the number of participants and the number of applications increase, this could lead to very complicated relationships. Also, it is important that the server as well as the client maintain this information in a clean fashion without mixing up any information with one another. We, therefore, invoke a new interpreter for each client that connects to the server. In addition, a new interpreter is also created for every application that is invoked by all the clients. Similarly, the client also invokes a new interpreter for every application that the user invokes. These interpreters are invoked using the command `interp create -safe` – we make use of *safe* interpreters to provide adequate security among the client/server

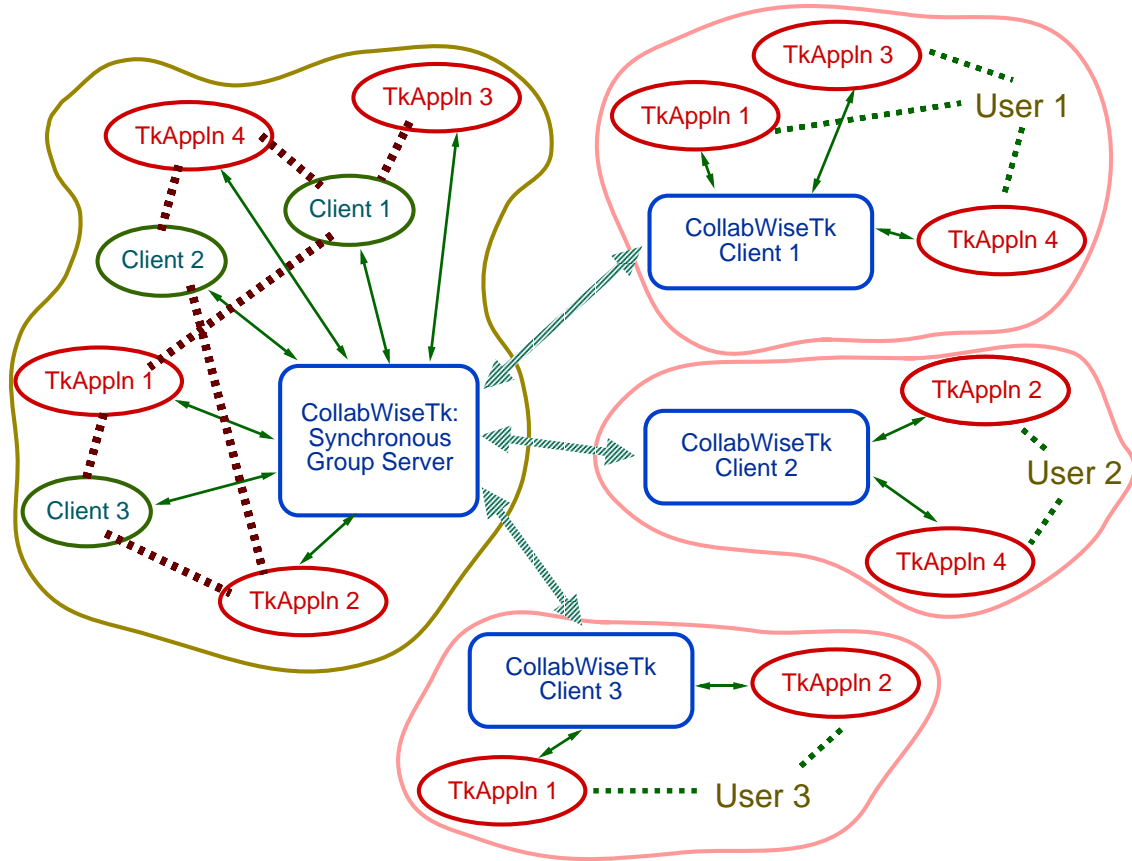


Fig. 4. Dependency relationships for a three-participant collaborative session in progress.

communication.

We next describe the mechanism used to capture events corresponding to user interactions and how these events are broadcast to its collaborative participants.

Interpreters on SGS are configured to provide a set of Tk commands such that in addition to being able to create, configure and delete generic widgets, these also include functionality to trap all the user interactions with the widget. Once the user interactions are trapped by the interpreter, it is upto sole discretion of the application-specific interpreter on how to process this information and it depends on the state of the inter-client synchronization table. For example, once the double-click on a button of TkAppln 1 by User 1 is captured, the interpreter does the following:

1. Check the current state of the button in inter-client synchronization table;
2. If the current state of the button happens to be in observe mode, then it immediately stops

processing the event any further;

3. If the current state of the button happens to be in interact mode, then it processes the corresponding event;

4. If any of the collaborative participants are setup in observe mode for this button, then the interpreter transmits the event to the SGS server for further processing.

SGC interpreters also perform the functionality to receive and process events that are transmitted by collaborative participants.

Client-specific interpreters on SGS are configured to:

1. receive events from their respective clients and forward it to the application-specific interpreter for broadcasting it to other relevant clients, and
2. relay or send the processed event from application-specific interpreters to their respective clients.

On the other hand, application-specific interpreters on SGS actually perform a inter-client synchroniza-

tion table lookup to process an event and send it to all the clients that are in observe state.

Since users are allowed to choose to choose and configure the state of a widget in an application, it is very easy for widgets to fall out of synchronization with corresponding widgets of collaborating participants. For example, this occurs when Alice changes the state of her text widget from `Interact/Alice` to `Observe/Bob`. Before Alice can start observing the Bob's interactions with his text widget, Alice has to first synchronize her text widget with that of Bob.

The client interpreter of Alice sends a request to SGS to retrieve information necessary for her text widget to synchronize with that of Bob. The SGS server in turn transmits the request to Bob's client interpreter and waits for an answer. The Bob's client interpreter processes this request and responds accordingly by providing all the appropriate information about the current state of its text widget. Once, the two text widgets are synchronized, Alice goes into observe mode and starts following Bob's interactions.

SGS and the collaborative client communicate with one another using a well-defined set of APIs. There are basically two types of API commands:

- synchronous API commands, which block the sequence of execution by waiting for the results; and
- asynchronous API commands, which do not block and wait for the results of the execution.

When client1 sends a user interaction to a collaborative participant client2, it is not necessary for the client1 to verify whether client2 has managed to receive this information or not. Hence, this falls into asynchronous type of command.

On the other hand, when client1 requests specific information about client2, such as the status of its scrollbar, then it is necessary to issue a synchronous command. However, this implies that synchronous commands will block the socket channel between the client-server and prevent communication of other commands. Therefore, we always send asynchronous commands over the socket channel, and make use of the 'vwait' command to implement the synchronous command.

Accordingly, we have developed two simple procedures to handle these two types of commands, as shown below:

```
##
# 1. Asynchronous transmission
#   sid = socket id
proc sendNforget {sid cmd} {
```

```
    transmitSocket $sid {} $cmd
};# End of proc sendNforget

##
# 2. Synchronous transmission
proc sendNreceive {sid cmd} {
    set returnId [clock clicks]
    transmitSocket $sid $returnId $cmd
    vwait $::returnId
};# End of proc sendNreceive
```

The procedure `sendNforget`, as the name suggests, sends the command across the socket channel and returns immediately. On the other hand, the procedure `sendNreceive` transmits a unique variable name, generated using the `clock` command and waits for this variable to be set using the `vwait` command. When this data is received on the other side, the presence of a variable name implies that a result is expected and therefore it transmits back the results such that this unique variable name is set to contain the results.

The notification of user interactions with a widget is achieved by analyzing the entire set of commands in tk widgets and re-defining these commands appropriately. Specifically, the original text widget is renamed and a new procedure is created by the same name. This new procedure performs actions that are necessary to:

- inform the group server, whenever a new text widget is created and if some other client is observing this text widget;
- create a new procedure for the text widget pathname that selectively sends similar information to group server.

The following tcl code snippet shows a simplified example of how a text widget can be made collaborative.

```
## Rename text widget
#
rename text text-Org
proc text {pathname args} {
    sendNforget $::sid "text $pathname $args"
    set ret [uplevel 1 text-Org $pathname $args]
    #rename $pathname cmd which just got created
    rename $pathname $pathname-Org
    proc $pathname {args} {
        # process $args
        switch -- $opt {
            cget {
                # send nothing to remote server, OR
                # use "sendNreceive $pathname cget"
```

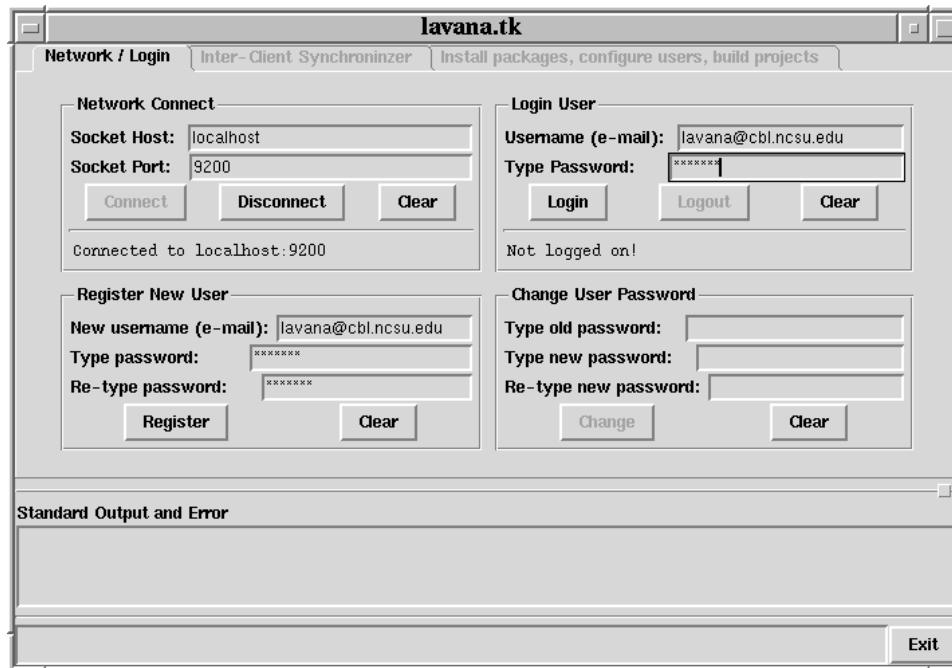


Fig. 5. Startup window of a CollabWiseTk client.

```

# cmd to get option value from rmt
# user, when configured for
# interaction with rmt user.
}
insert {
  sendNforget $::sid \
    "$pathname insert char"
}
};# End of switch stmt
# ....
};# End of proc $pathname
return $ret
};# End of proc text

```

Whenever the application invokes a text widget, the above procedure is called. It first informs SGS about the creation of the text widget using the non-blocking command `sendNforget`. It then proceeds with the creation of the actual text widget. Since creation of a text also creates a new command by its widget name called `pathname`, it re-defines `pathname` to `pathname-org` and creates a wrapper script for it. Other Tk widgets are implemented similarly.

6 Testbed and Experiments

We decided to use the Tk widget demonstrations, distributed with the core Tcl/Tk, as a test-bed for testing the CollabWiseTk toolkit. We have chosen these demos because they not only cover most of the commands in the tk widget set, but also demon-

strate usefulness of the toolkit in rendering these applications collaborative. The experiments need to be conducted in two phases:

- manually invoke all the demos and verify their operation in several different configuration modes; and
- setup a testbed and conduct a series of experiments to evaluate the performance and scalability of this architecture.

Our current implementation of the toolkit allows us to collaborate many of the listed demos. However, we have not yet implemented: (1) one major tk widget, namely the canvas widget; and (2) the tagged items on the text/canvas widget in our collaboration toolkit. Figure 5 shows the initial startup window of a collaborative client, which allows the user to connect and login to the synchronous group server. Figure 6 shows the main window of the tk widget demonstrations invoked in collaborative mode. Specifically, it shows a 15-puzzle game which has been made collaborative for two players as described next. First player can only click on the odd buttons whereas the second player can only click on the even buttons. Two kinds of games can be played with such a configuration: (1) players assists one another in completing the game at the earliest; or (2) one player tries to prevent the other player from completing the game.

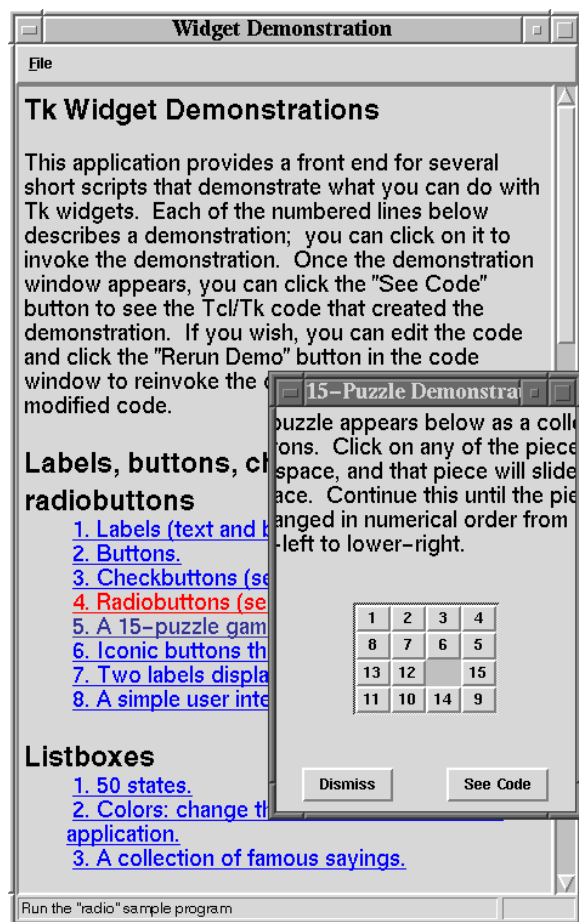


Fig. 6. Collaborative Tk widget demos.

7 Software Evaluation

We evaluate the *CollabWiseTk* toolkit in the context of several factors below.

User configurability. We provide the flexibility to save the mode of collaboration for a specific application in a static file. A programmer can thus anticipate a variety of useful collaboration modes and save it in separate configuration files. Users can then invoke the application with appropriate configuration file during collaborative sessions. However, users are not limited to using the collaborative mode defined in these files, but have the flexibility to also change the mode of collaboration, during run-time, to suit their specific needs.

System architecture. We have chosen a hybrid architecture to implement *CollabWiseTk*. A replica of the single-user application executes on every user's machine thereby providing good response times for local interactions. On the other hand, a centralized server is used for synchronization and maintain-

ing consistent state information. This may result in performance bottlenecks where high interactivity is needed for increased number of participants.

Group awareness. Inter-client synchronization table maintains a list of active sessions corresponding to each application being shared. Additionally, every session not only lists the number of users actively collaborating on an application, but also displays the type of user activity such as interact or observe for a specific widget. This enables other users to determine the status of a particular user in respect to a specific widget and aids in increasing the group awareness significantly.

Floor control. The toolkit provides a very fine granularity over floor-control and allows the users to configure the interaction mode down to the widget element, instead of merely allowing control of the entire application. Additionally, users also have the capability to dynamically change and allow other users to interact with a specific widget of choice whenever the need arises. This provides better flexibility in letting users drive the mode of collaboration to suit their needs, rather than a programmer trying to anticipate all the collaborative needs for design specific collaborative-aware application.

Scalability. The scalability of the toolkit depends on the users and how they decide to share a specific session during runtime. If users want to be aware of all the activities of the other users, then there will naturally be a performance hit when the number of users increase. However, it is expected that most users will not work in such fully shared mode and would prefer sharing only part of the widgets from the entire application. This would minimize the communication overhead and hence this toolkit would offer good scalability even when the number of users increase.

Limitations. The architecture relies on a single centralized server to share the information for collaboration. This can result in disruption of collaboration services if the server fails for any reason.

8 Software Availability and Status

The *CollabWiseTk* toolkit has been currently implemented for most of the tk widgets, except canvas. We also plan to make the toolkit available on the Web, once the canvas widget is fully implemented and we go through in-house testing phase.

Further details about the current status of these packages will be made available under:

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

9 Conclusions

We have demonstrated the capabilities of the `CollabWiseTk` toolkit by readily transforming most of the existing Tk Widget Demos into collaborative applications. Furthermore, the versatility of the toolkit is realized by the flexibility that it provides in rendering a stand-alone application into a variety of collaborative modes with little amount of work. Additionally, the functionality of these collaborative clients can be dynamically re-configured by the participants - thereby making it a very useful toolkit.

The re-configurability of the low-level primitives can be combined with one another to form several useful mega-widgets that better reflect users's models of work. This is illustrated with the help of a text widget and a vertical scrollbar, which is rendered collaborative as

- a shared editor,
- a chat box window, or
- a local text box supporting remote edits.

Similarly, other primitive widgets can be combined to create numerous useful applications.

References

- [1] F. Brglez, H. Lavana, Z. Fu, D. Ghosh, L. I. Moffitt, S. Nelson, J. M. Smith, and J. Zhou. Collaborative Client-Server Architectures in Tcl/Tk: A Class Project Experiment and Experience, February 2000. Seventh Annual Tcl/Tk Conference, Feb 14-18, 2000, Austin, Texas.
- [2] F. Brglez. Frontiers of Collaborative Computing on the Internet, A Graduate Course Experiment, January 1999. Two project reports, published after the completion of the course, are also available from the course home page under <http://www.cbl.ncsu.edu/~brglez/csc591b/>.
- [3] M. Harrison and M. McLennan. *Effective Tcl/Tk Programming*. Addison-Wesley, 1998.
- [4] GroupKit Version 5.1. Published under URL <http://www.cpsc.ucalgary.ca/grouplab/groupkit>, 1998.
- [5] TANGO: Collaboratory for the Web. Published under URL <http://trurl.npac.syr.edu/tango>, 1998.
- [6] G. Konduri and A. Chandrakasan. A Framework for Collaborative and Distributed Web-Based Design. In *Proceedings of the 36th Design Automation Conference*, June 1999.
- [7] 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>.
- [8] 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>.
- [9] A. Khetawat, H. Lavana, and F. Brglez. Internet-based Desktops in Tcl/Tk: Collaborative and Recordable. In *Sixth Annual Tcl/Tk Conference*. USENIX, September 1998. Also available at <http://www.cbl.ncsu.edu/publications/#1998-TclTk-Khetawat>.
- [10] S. Greenberg and M. Roseman. Groupware Toolkits for Synchronous Work. In M. Beaudouin-Lafon, editor, *Computer-Supported Cooperative Work, Trends in Software Series*. John Wiley & Sons Ltd., 1998. Also available as a Research Report 96/589/09, Dept. of Computer Science, University of Calgary, Calgary, Canada, under <http://www.cpsc.ucalgary.ca/projects/grouplab/papers/1998/98-GroupwareToolkits.Wiley/Report96-589-09/report96-589-09.pdf>.
- [11] S. Greenberg. Real Time Distributed Collaboration. In P. Dasgupta and J. E. Urban, editor, *Encyclopedia of Distributed Computing*. Kluwer Academic Publishers, 1999. Also available as a Research Report 96/589/09, Dept. of Computer Science, University of Calgary, Calgary, Canada, under <http://www.cpsc.ucalgary.ca/projects/grouplab/papers/1998/98-Encyclopedia-Distrib/encyclopedia-realtime-collaboration.pdf>.
- [12] BWidget Toolkit. Published under URL <http://www.unifix-online.com/BWidget>, 1999.