

Collaborative Client-Server Architectures in Tcl/Tk: A Class Project Experiment and Experience

Franc Brglez Hemang Lavana
Zhi Fu Debabrata Ghosh Lorie I. Moffitt
Steve Nelson J. Marshall Smith Jun Zhou

Dept. of Computer Science, Box 8206
NC State University, Raleigh, NC 27695, USA
<http://www.cbl.ncsu.edu/~brglez/csc591b/>

Abstract

This paper presents a class software project that was part of a recent experimental graduate course on Frontiers of Collaborative Computing on the Internet. We chose Tcl/Tk to facilitate rapid prototyping, testing, and demonstrating all phases of the project. The major milestones achieved during this course are:

- *rapid proficiency in Tcl/Tk that allowed each student to manipulate data and widgets, apply socket programming principles, and create a progression of client/server applications, from textbook cases to a unique client/server architecture prototype – driven by and matched to a well-defined collaborative project driver.*

- *universal server that supports any number of user-configurable clients, each accessible through a Web-browser on a Mac, Windows, or UNIX platforms. Prototype client configurations include: (1) collaborative document composition, (2) collaborative Tcl/Tk debugging and compilation, (3) collaborative design workflow.*

Keywords: client-server architectures, collaborative computing, Internet, Tcl/Tk, GUI.

1 Introduction

There is no consensus about the definition of *collaborative computing*. It appears that while *Computer-Supported Cooperative Work* (CSCW since 1960's) may have been eclipsed by *groupware*, the notion of collaborative computing is still being molded. Checking out a web-based search engine with keywords such as 'CSCW' returns about 44,470 links to related Web pages, 'groupware' returns about 228,940 links (mostly about Lotus Notes, NetWare,

Office97, ...), while 'collaborative computing' returns only 8,857 links, several already listed under earlier keywords.

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]). This course defines 'collaborative computing' as the hardware, software and structures that support a group of individuals working on related tasks – where hardware, software, data, and individuals may be distributed over a wide geographical area. 'Groupware' is a subset of collaborative computing: it is the software that allows communication, coordination and the sharing of information between distributed individuals and groups. Common examples of groupware are e-mail and video conferencing facilities, shared access to databases of documents and images, and applications such as shared white boards. A number of commercial systems are available to support such activities and the systems continue to evolve [3].

The main goal of csc591-b is to introduce collaborative computing as a distributed process, asynchronous and synchronous, that invokes, links, and executes

- data sets residing/generated on local/remote hosts;
- heterogeneous applications residing on local/remote hosts;
- floor control for collaborating teams distributed among local/remote hosts.

To make this process less abstract, we introduced the notion of a collaborative project:

An entity with distributed participants, distributed data sets and libraries, distributed tool sets and libraries, and objectives to be met by completing well-defined sequences of tasks – assigned by the the project leader, and subject to collaborative project activity and reviews.

* F. Brglez and H. Lavana have been supported by contracts from the DARPA/ARO (P-3316-EL/DAAH04-94-G-2080 and DAAG55-97-1-0345), and a grant from Semiconductor Research Corporation.

A project driver that matches this definition, and is within the scope of a single semester graduate class, is the creation of a distributed environment that supports collaborative document composition. The class instructor acts as a project leader and document editor, while students, having created a distributed environment first, maintain bibliographical databases, create graphics for embedded illustrations, write and compile the pre-assigned sections of the document. Most of such activity uses clients that interact with the local host of each participant, while periodic synchronization and sharing of data with other participants takes place when each client links to the common server.

While the environment as described above appears specific to document composition, the implementation, conceived as part of the class project, demonstrates a universal client/server architecture where the same server can now support any number of user-configurable, project-specific tasks or workflows. In order to complete most of the key objectives of the project before the end of the semester, we made Tcl/Tk [4] the scripting language of choice at the very beginning. The first few weeks of the course were devoted exclusively to the introduction and exercises in Tcl/Tk.

Our choice of Tcl/Tk for the class project in collaborative computing was also influenced by a number of favorable experiences with Tcl/Tk, each of which could also be shared and critiqued in the class environment, such as GroupKit [5], AgentTcl [6], user-configurable workflows [7], recording and playback toolkit [8], and toolkit for the web-browsers [9]. This paper introduces the *Asynchronous Group Server Architecture* (AGS) and a platform-independent client as it was conceived and implemented for a number of collaborative demos as part of the project in this course. The paper is organized into several sections as follows:

- Course Organization;
- Collaborative Project Driver;
- GUI and Client/Server Architecture;
- Server Design;
- Client Design;
- Collaborative Experiments;
- Conclusions.

A complementary architecture that also evolved from this class project, in particular the *Synchronizing Group Server Architecture* (SGS) and the transformation of single-user client-applications into collaborating client-applications is presented in a companion paper [1].

2 Course Organization

The class brought together a team of seven students, each with some programming experience under UNIX, WindowsNT, and MacOS. Few students had limited experience in Java programming, and only one student had significant experience in Tcl/Tk programming. The remaining students have learned about the rudiments of Tcl/Tk programming during the first few weeks of the course so that the remainder of the course could be devoted to a major joint class project that applied and extended the principles of client/server programming under Tcl/Tk. Specifically, the course was organized into three major sections:

- Rudiments of Tcl/Tk, including socket programming and client-server architectures;
- Case studies of various client-server architectures;
- Collaborative project outline, implementation hints, implementation reviews, final presentation and report.

The nominal textbook for the course was *Effective Tcl/Tk Programming* [10]. Chapters that were covered in some detail include Event Handling, Canvas Widget, Text Widget, Top-level Windows, and Interacting with Other Programs. Preceding the introduction of this textbook was a series of lectures on Tcl, mostly based on the material from [11, 12]. Tcl/Tk man pages and related links were made available on the course home-page [2].

Hands-on case studies, in class and as homework, of various client-server architectures covered *The Electric Secretary* in [10], *GroupKit5.1* in [5], *REUBEN* in [7]. In addition, the class was introduced to *Web-WiseTclTk* in [9], which assisted in migrating the client application to the Web. Some of the homework assignments were completed and submitted as recorded sessions, using *RecordnPlay* in [8]. The introduction of AgentTcl in [6] was dropped due to lack of time.

A number of collaborative client/server architectures are known. Principally, they deal with single applications rather than workflows of applications, ranging from a shared calendar (The Electric Secretary) [10] to a shared whiteboard [5]; from collaborative visualization for health care [13] to collaborative editing of schematic diagrams [14]. This class project builds on the recent experiences with collaborative workflows of heterogeneous applications [7, 8, 15] and the client-server architecture of *The Electric Secretary* in [10]. The latter served as the initial model for the *Asynchronous Group Server Architecture* (AGS) described in sections that follow.

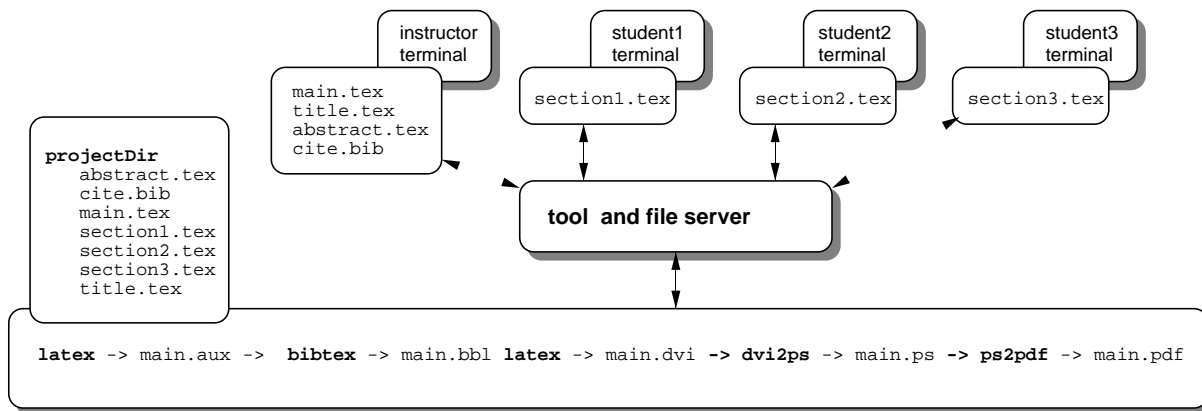


Fig. 1. An informal arrangement to collaboratively execute a document composition project.

3 A Collaborative Project Driver

The example that motivated and guided much of the class discussion and one that became the project driver, is first shown as an informal arrangement in Figure 1. The project involves the project leader (instructor) and a number of students. The objective of the project is to write a joint paper on the technology that will have enabled the class to devise and implement a client/server architecture to compile the paper in a collaborative mode. In the arrangement as shown in Figure 1, we assume

- all students have login access to a common tool and file server,
- each student will use a unique file name when writing a section assigned by the project leader,
- each student can create her own ‘main’ program that includes her section, and may be sections from others, to compile into a composite document that can be viewed or printed as a postscript or a pdf document by anybody in the class.

The compilation process itself may consist of all or some of the following steps:

1. first execution of `latex` [16] on file `main.tex` to output a file `main.aux`;
2. execution of `bibtex` on file `main.aux` to output a file `main.bbl` (to be cross-referenced with citations contained in a file `*.bib`);
3. second execution of `latex` on file `main.bbl` to output a file `main.dvi` (now containing the full document);
4. execution of `dvi2ps` on file `main.dvi` to output a file `main.ps` (ready for printing or viewing as a postscript file);
5. execution of `ps2pdf` on file `main.pdf` to output a file `main.pdf` (ready for printing or viewing as a pdf file).

This arrangement, while workable in principle, has a number of drawbacks:

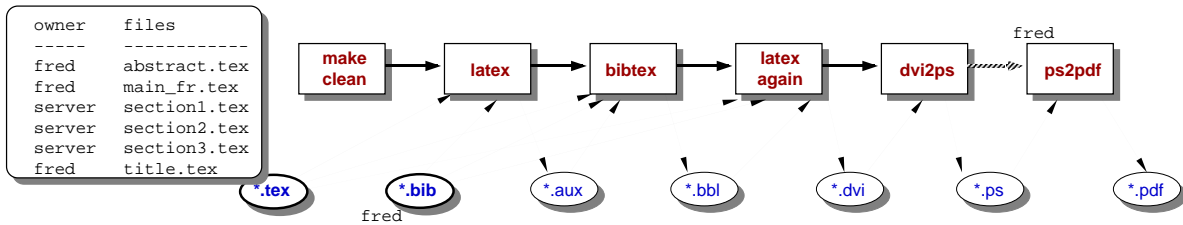
- it requires all participants to have a login account on the server;
- it is based on ad hoc coordination of files and symbolic links in the executable directory (owned by the project leader) versus the data files that are written by students in their home directories for which a symbolic link must be maintained in the executable directory by the project leader.
- it would require `ftp` in order for students to maintain their files on their local client hosts;
- it has no structure to render editing of files, or execution of the compilation sequence interactive and collaborative.

We next introduce an abstraction that formalizes the project description in Figure 1 such that we can also render it collaborative.

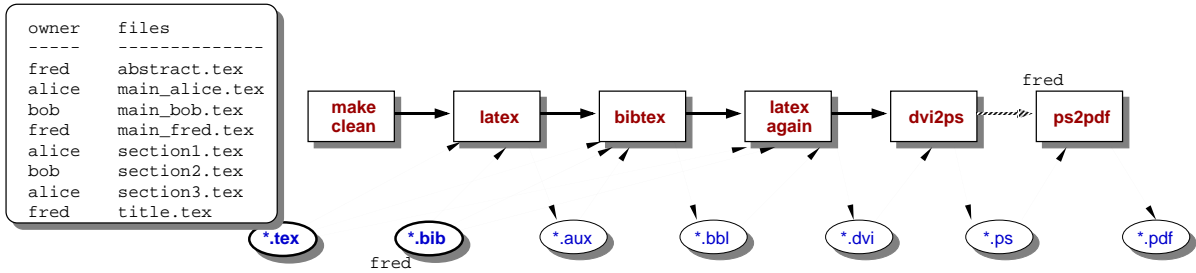
Formalizing the views. The document composition project illustrated in Figure 1 can be executed using a `makefile` – but only if data and tools reside on the same host. In general, this is not the case. The sequence of executable tasks in Figure 1 defines a simple workflow that can be represented with an acyclic Petri net graph consisting of data and tool nodes, with tool nodes acting as transitions that execute (or ‘fire’) only when all required input data is present at the inputs of the nodes. This is a special case of the more general case that involves graphs with cycles [7].

The graph representation, with distributed data nodes invoking tool nodes on any number of distributed file servers, is an effective GUI for the workflow client applications. In particular, such representation can readily support collaborative features and user-configurability. Three views of the essential features of such a client interface are outlined in

(a) Initial configuration of the project-specific workflow client (as initiated by project leader Fred)



(b) Asynchronous configuration of the project-specific workflow client (as continued by project team)



(c) Synchronizing configuration of the project-specific workflow client (as completed by project team and the leader)

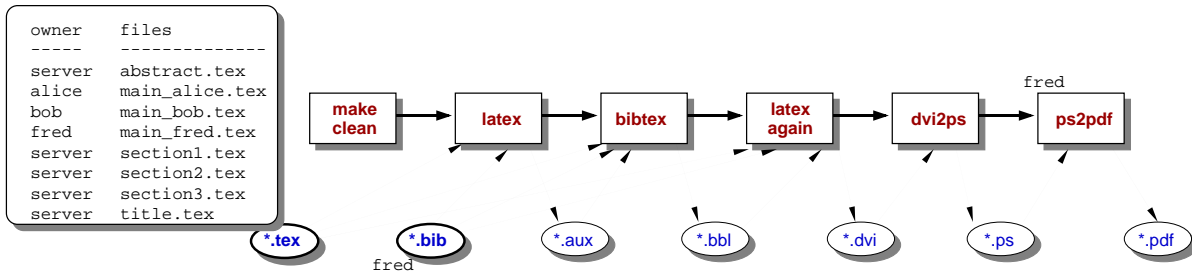


Fig. 2. Three views of a project-specific client workflow for collaborative edits and execution.

Figure 2: *set-up view*, *asynchronous view*, and *synchronizing view*.

Without loss of generality, we continue using the document composition project illustrated in Figure 1 as the illustrative example. To keep the presentation simple, the project leader (Fred) engages only two participants: Alice and Bob. The expected output from this project is a multi-section document on a specific topic, with Alice and Bob contributing technical sections, and Fred acting as the editor and also contributing the cover section, the abstract section, and the maintenance of the bibliography database.

Set-up View. Shown in Figure 2(a), this is a view of the client as initially configured by the project leader (Fred). In this project, task completion implies sequential execution of one or more tools from the set $\{makeClean, latex, bibtex, dvi2ps, ps2pdf\}$. With the exception of *makeClean*, these tools read instances of input files and write instances of output

files. The task of the tool *makeClean* is to delete all of the intermediate file classes such as $\{*.aux, *.bbl, *.dvi, *.ps, *.pdf\}$ since any file instances in these classes can be regenerated by invoking the tasks that are driven by instances from *primary input* file classes such as $\{*.tex, *.bib\}$. Generating and editing instances of files in these two classes represents the essential contribution to the project from each member of the team.

In the set-up view, Fred has the exclusive r/w ownership of all files in the class **.bib* as well as the files ‘main_fred.tex’, ‘title.tex’ and ‘abstract.tex’ in the in the class **.tex*. The files ‘section1.tex’, ‘section2.tex’, and ‘section3.tex’ are shown as owned by ‘server’, indicating that other team members are free to claim ownership to any of them. The file ‘main_fred.tex’ has a listing of any **.tex* and **.bib* files that are to be included in the compiled document. With the exception of *ps2pdf*, shown as con-

trolled exclusively by Fred, there are no restrictions on execution of other tools. At this point, Fred can execute and test the task execution of the entire workflow by invoking *makeClean* and then *latex* on ‘main_fred.tex’, to be followed by other tool invocations until reaching *ps2pdf*. Alternatively, any segment of the tools could be chained for automated execution of the entire task sequence or any subsequence. At this stage, the files ‘section1.tex’, ‘section2.tex’, and ‘section3.tex’ are simple template files with tentative titles and are yet to be expanded and edited by Alice and Bob.

Asynchronous View. Shown in Figure 2(b), this view of the client depicts the work in progress, with contributions from Fred, Alice and Bob. Specifically, Alice claims ownership of ‘section1.tex’, and ‘section3.tex’, while Bob claims ownership of ‘section2.tex’. In addition, each has created files ‘main_alice.tex’ and ‘main_bob.tex’, respectively, to execute any desired combination of the files from *.tex and *.bib class. In this view, the three participants work to a large extent independently, while each can always access, read, and process the files created by others. Typically, each may be editing the files on a local host rather than the server where all files are accessible to all the tools in the flow.

Synchronizing View. Shown in Figure 2(c), this view depicts the state of the client when the project leader Fred has scheduled a project review and a collaborative editing session in which all team members participate. While the team may reside at different locations, all communicate with each other, at the minimum via a chat-like window on the terminal screen. In the best case scenario, participants may communicate also via audio and/or a video channels. The main goals of the review are (1) to synchronize versions of input files generated by distributed team, (2) to review and edit individual files, and (3) to share the control of the flow execution for the final version of the document. This is an interactive and collaborative process in real time.

In the synchronizing view as shown in Figure 2(c), most files are returned to the ownership of the server – allowing any team member to access them in r/w mode for editing. In particular, we note that Bob has secured access to ‘abstract.tex’ – a file originally generated by Fred. The decision to access this file by Bob has been made after a brief discussion among all team members. In its crudest form, the mechanism by which others can observe Bob’s editing is to download the revised file on prompts from Bob. In a more elaborate environment, others may observe Bob’s editing in real time on their own termi-

nal screen. Ultimately, one may expect an environment where two or more members may be editing the same file in real time in a user-friendly and an unambiguous manner.

Issues in rendering a client collaborative.

Given the code for the stand-alone client application, 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 [5, 17, 18]. Such issues are addressed, and an effective solution proposed, in the companion paper [1].

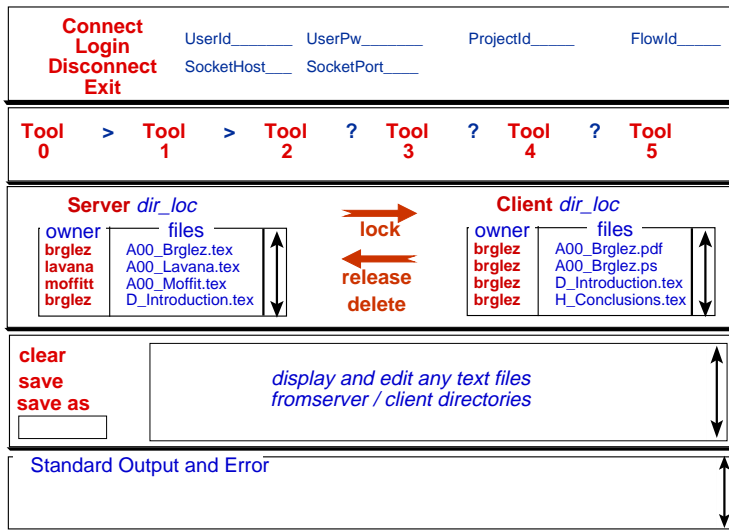
Generic examples of issues in rendering a stand-alone client collaborative, as introduced and discussed in the class setting, are included in the lecture notes [2] and the companion technical report [19]. The project driver example as introduced in Figure 1, and more formally in Figure 2, has been instrumental in arriving at the two-way partition of the server architecture to support two dynamic views of a collaborative project:

1. asynchronous view, defining parameters for an *asynchronous group server* (AGS);
2. synchronizing view, defining parameters for an *synchronizing group server* (SGS).

Due to limited time, the emphasis of the course was to prototype a client/server architecture that primarily supports the asynchronous view of collaboration and is described in the remainder of this paper. However, the simple demos, by the end of the class, that demonstrated the feasibility and the potential of the synchronizing view of collaboration, have also provided the direction beyond the class setting, leading to the companion paper on the SGS client/server architecture and implementation [1].

4 GUI and C/S Architecture

We have defined the concept of a collaborative project and the notion of a partitioning a set of project-specific tasks and data among project participants – giving rise to one or more workflow clients that are to be executed by participants in asynchronous and synchronizing modes. In addition, we identified behavioral classification of objects in the workflow client partitions, impacting the way we propose to implement a collaborative client. All of



The workflow toolset is user-configurable and invoked by a specific *FlowId*, whereas *UserId* and *UserPw* are required to access a project-specific directory identified by *ProjectId*. Before clicking on the **Connect** button, user enters *SocketHost* and *SocketPort*. After clicking on the **Login** button, files in the project-specific directory are listed in the respective Server and Client widgets. Participants can release ownership of files by clicking on the files they own. File transfers between client and server directories can be executed with or without locking the ownership.

User-invoked tools are displayed as a tool chain and users can enable/disable any links by clicking on the > or ?. In this example, clicking on Tool0 invokes the tool, which on completion will also invoke Tool1, followed by Tool2.

Fig. 3. An example of a universal user-configurable workflow client interface.

these factors have influenced the current view of the universal user-configurable workflow client interface and the corresponding server architecture.

We first specify the GUI of the client, and describe how it relates to the proposed client/server architecture. This specification served as a blueprint for the student teams implementing several versions of both the server (in Tcl) and the client (in Tcl/Tk), as described in the subsequent sections.

GUI for the Client. The currently proposed and implemented workflow client interface consists of five major ‘frames’, each with a number of functions:

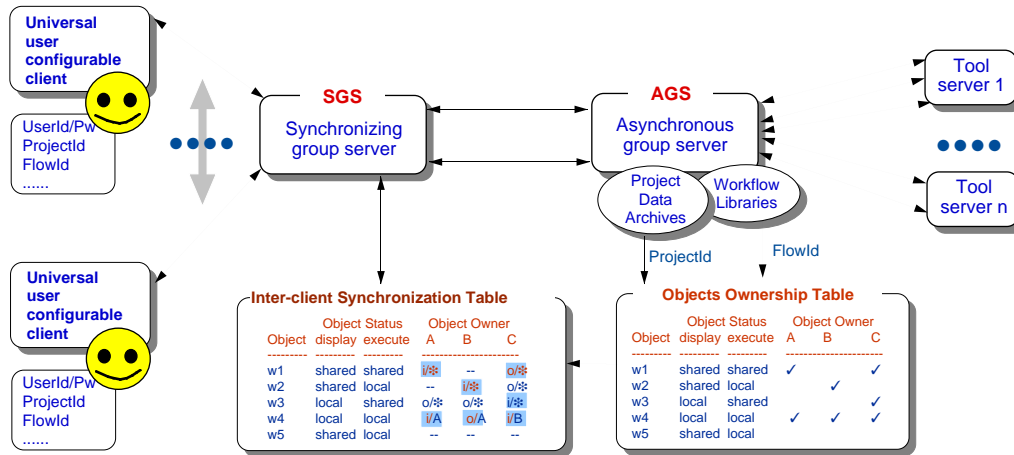
1. **loginFrame** supporting
 - buttons to *connect*, *login*, *disconnect*, *exit*
 - entries for *userId*, *userPw*, *projectId*, *flowId*, *socketHost*, *socketPort*
2. **toolFrame** supporting
 - buttons to invoke *tool0*, *tool1*, *tool2*, ... The bindings for the tools may be hard-wired in the initial versions of the client, but will be loaded later from a user defined configuration file, identified with a specific *flowID*.
 - connectors to connect/disconnect successive invocations of *tool0*, *tool1*, *tool2*, ...
3. **filesFrame** supporting
 - server button that may toggle and display either server top-level project directory location or participant’s subdirectory location on the server (the latter is created from project partitions by the project leader).
 - two-column listbox or textbox listing file owner and file name in the server directory or sub-

directory.

- client button that may toggle and display either local client project directory location or the interacting participant’s local directory (subject to mutually agreed permissions).
- two-column listbox or textbox listing file owner and file name in the respective client directory.
- entries for buttons to *lock/release* ownership of selected files, button to *delete* a selected file, arrow buttons to *upload/download* a file to/from the server.
- 4. **editFrame** supporting
 - buttons to *clear*, *save*, *save as* files brought into the display and edit window.
 - text widget to display and edit any text file from the server or client directory (after clicking on the selected file). All files can be accessed for display, file owners can edit them.
- 5. **stdoutFrame** supporting
 - text widget to display any messages directed to standard output or to standard error.

A sketch of the proposed GUI for this client is shown in Figure 3, along with illustrative text of representative user interactions.

Client/Server Architecture. The proposed client/server architecture in Figure 4 matches the concept of a collaborative project as stated earlier: a number of workflows configured and partitioned by the project leader may be associated with each project; whereas participants may work in an asynchronous mode on the project partitions as well as



A number of workflows may be selected for each project, the objects in each workflow may be assigned a project and a team specific *object ownership table* by the project leader. This static table initializes the *Inter-client synchronization table* where activity permissions related to the objects can be changed dynamically by the participants, subject to the initialization constraints. For example, participant *C* can interact with object *w3* and share the interactions with all participants (*i/**), while participants *A* and *B* can only choose to observe (*o/**) or not observe (*-*) the same object. However, since all participants are initialized as owners of *w4* (the shaded entries in this table correspond to the permission checkmarks in the initialization table), all can choose to interact in variety of ways with this object: *A* can interact with *w4* alone, (*i/A*), with only participant *B* observing (*o/A* under the column *B*); *C* can interact with *B* only (*i/B* under the column *C*). Each participant can click on the entries in this table and 'toggle' the entry into the desired or allowable state. For example, the object *w3* can be assigned to *C* as *i/**, *o/**, or *-*, while the same object can be assigned to *B* only as *o/** or *-*.

Fig. 4. The SGS/AGS client-server context and architectural features.

a synchronizing mode. This concept is reflected in the client/server architecture which itself is partitioned into an Asynchronous Group Server (AGS) and a Synchronous Group Server (SGS). For each project and workflow invoked by the participant, the AGS maintains not only the project data archives and workflow libraries but also an *Objects Ownership Table* preconfigured by the project leader. Once invoked, the *Objects Ownership Table* initializes the *Inter-client Synchronization Table* which interacts with SGS and the participants' clients.

A brief description of anticipated user-interactions in the interactive collaborative mode (where the *Inter-client Synchronization Table* can be changed dynamically by the project participants) is given in Figure 4. See the companion paper [1] for more details.

Project status by the end of semester. One student has implemented a version of the AGS server that supports all features as defined for the client in Figure 3 – except the file subdirectories for individual participants. A total of five similar but different clients have been independently developed by collaborating pairs of students. The *toolFrame* in one of these clients is user-configurable, so the client is truly universal.

In the document composition project, each partici-

pant can maintain/release ownership of files. A simple naming convention allows each member to execute the project partition independently of others, while the project leader can assemble and edit the complete document for immediate access and feedback to all. The client is invoked through a Web browser from any platform (UNIX, WindowsNT, MacOS) and collaborative execution can take place with distributed participants.

The paper concludes with highlights of AGS design, client design, and collaborative experiments conducted with the universal client/server architecture for three different applications: distributed document composition, distributed software debugging and compilation, and distributed experimental design environment.

5 Server Design

The design of a collaborative server was very critical to the success of this class project. There are several requirements of the server. It should support and maintain:

- connections from different students, after verification of their identity;
- several different projects and workflows accessible by the students;

- restriction for each project to access only those specific tools that are required by the project;
- ownerships of data files in the project directory;
- data transfers between the server's project directory and the student's client host;
- invocation of tools in each project directory.

The first step was to define and develop a communication API (application programmer's interface) that can be used by the client/server to meet design requirements, as outlined above. We decided to employ *asynchronous communication* scheme, as described in [10]. The server was configured to understand a minimal set of commands. The mechanism for client/server communication is described next. The client sends a request to the server which consists of a string of the following form:

```
server_cmd arg1 arg2 ... argn client_cmd
```

The server processes the request from the client by invoking a procedure `server_cmd`, in a safe-`interpreter`, with n arguments. After the `server_cmd` completes its execution, the server parses the `client_cmd` string and replaces all occurrences of (1) "%v" with the result returned by `server_cmd` execution, and (2) "%l" with the length of the result returned by `server_cmd` execution. The new `client_cmd` string is then sent back to the client, which processes the string received in a safe-`interpreter`. However, whenever execution of `server_cmd` results in an error, the server sends `error_result reason` to the client, instead of sending back the `client_cmd` string. The list of commands recognized by the server falls into four categories:

1. *Initial set-up/login process:* Once the clients establish a socket connection to the server, it is necessary to identify the user using the command:

```
login <user> <pswd> <project> <client_cmd>
```

If the user logs in successfully, then the following commands are made available to the client.

2. *Ownership of data:* The following commands allow the client to perform ownership related operations on a data file:

```
checkOwnership <filename> <client_cmd>
grabOwnership <filename> <client_cmd>
releaseOwnership <filename> <client_cmd>
```

A client can assume ownership of a data file, only if it is owned by the server. On releasing the ownership of a data file, the server becomes its owner. All three commands return the current ownership of the file.

3. *Data transfer:* Clients can get the list of data files, download, upload or delete a data file, as

follows:

```
getList <client_cmd>
downloadFile <file> <client_cmd>
uploadFile <file> <data> <client_cmd>
deleteFile <file> <client_cmd>
```

4. *Tool invocation:* A client may invoke a tool on the server using the following command:

```
executeCommand <what args> <client_cmd>
```

This single command gives the server the flexibility to invoke any tool specified in `what args` without having to restart the server when a new tool need to be added to the services. However, it can also be major concern for security. We resolve this issue as described next.

Configuration Makefile. The server is designed to maintain several projects. Each project is restricted to access a limited set of tools only, depending on what tools are required by the project. A configuration `makefile`, stored in the project directory, determines the set of tools available for any given project. Each tool needs to be explicitly specified in the `makefile` along with the command line arguments necessary for its invocation. A typical entry in the `makefile`, which generates a pdf file from a postscript document, is as shown below:

```
document=A00_main
ps2pdf:
    ps2pdf $(document).ps $(document).pdf
```

This task may be invoked from the command line as `make ps2pdf` or `make ps2pdf document=A00_Brglez`. In the first case, the default value (A00_main) of the document is used, whereas in the second case, the document rootname is specified on the command line. Thus, a client would typically send the following command to the server to invoke the above task:

```
executeCommand "ps2pdf document=A00_Brglez" "puts {%v}"
```

The server will invoke this task, if available, in the project directory.

6 Client Design

As the course evolved, so did the versions of the client implementations and likewise, the versions of the server implementations – a true learning experience for everyone in the class. Unlike the server design which was completed by a single student who had prior experience with TclTk, the client design was completed by students who only learned TclTk during this course. While sharing the debugging experiences with each other, a total of five clients were designed independently by five student teams.

Each of the clients implemented the basic function-

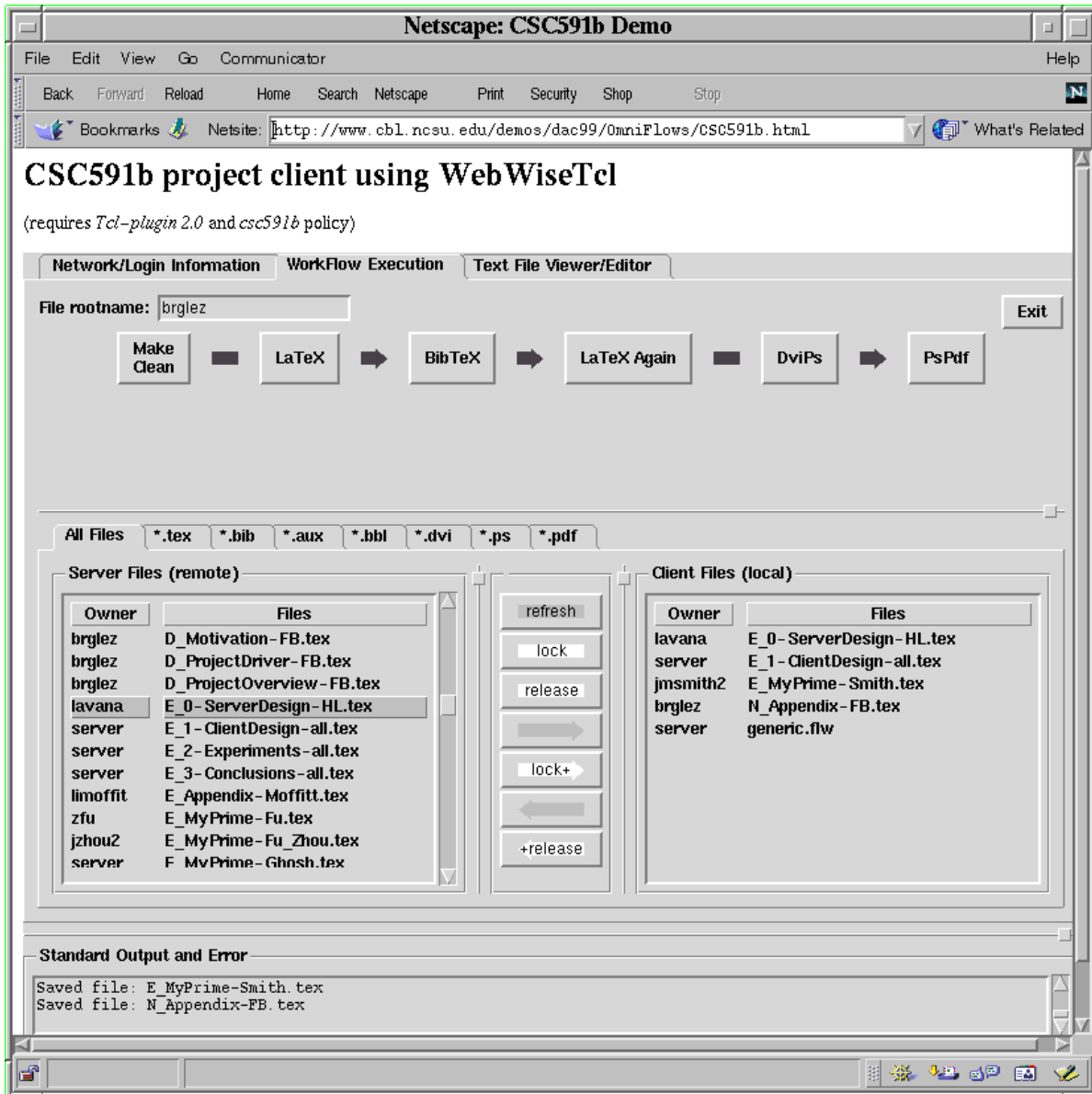


Fig. 5. Client design 1 (Here, the *toolFrame* is loaded from a configuration file).

ality as specified in in Figure 3 and the screenshot of each client, *demonstrated and tested for collaborative document composition functionality during the last session of the class*, is archived as part of the report posted on the class home page [2]. Each of the clients was tested through a Netscape browser and was executable from a UNIX, MacOS, and WindowsNT workstation, provided the browser has installed the WebWiseTcl [9], Safe-Tcl plugin, and the csc591b (course) policy. The latter is required to download and save the files from the server to the local client.

Each team was also responsible to contribute a subsection highlighting elements of TclTk used to

achieve the required functionality. A representative GUI for the client is shown in Figure 5. Like all other clients designed in this course, this client is executable from window in a web-browser. This particular client is universal: the contents of the *toolFrame* are loaded from a user-specified configuration file, which is linked to the project-specific makefile maintained by the server. In addition, this client makes use of the B-widget Toolkit [20] to implement its GUI.

We use the the initial client interface specification in Figure 3 to briefly describe the GUI of the client in

Figure 5.

loginFrame. The loginFrame is invoked by clicking on ‘Network Information’ in Figure 5. A set of entry boxes will prompt the user to enter ‘UserId’, ‘UserPw’, ‘ProjectId’, ‘FlowId’, etc. A set of well-placed buttons will allow user to ‘connect’, ‘login’, ‘disconnect’, ‘loadFlow’, ‘removeFlow’, and ‘exit’.

toolFrame and reconfigurability. The toolFrame can be ‘hard-wired’ as part of the flow-specific client itself (e.g. the composition project flow), or can be loaded as a user-specified configuration file that is linked to the flow-specific *makefile* on the server. This configuration file can only contain a subset of the targets and dependencies of the makefile. The toolFrame of the client shown in Figure 5 has been generated automatically by loading the configuration file. The flow shown chains the applications that are used in the document composition project and can invoke ‘Clean’ to remove old work files, ‘LaTeX’ to generate a compiled version of the document, ‘BibTeX’ to create citation indices, ‘LaTeX Again’ to load to update the document with citation indices, ‘Dvi2ps’ to create a postscript file, and ‘Ps2pdf’ to create a document in the pdf format. In the example shown, links between ‘LaTeX’, ‘BibTeX’, ‘LaTeX Again’ have been activated by the user (shown now as arrows after each click on the connector bar), so all buttons are executed consecutively once the user clicks on ‘LaTeX’. Such control of execution, from task i to task j , cannot be achieved with the make utility, where only the end-task j can be specified by the user.

The entry for ‘file rootname’ allows each user to select the name of the ‘main.tex’ file that should be invoked upon execution of the flow. Additional examples of the universal client invocation, for different sets of tasks, are shown in Figure 6.

filesFrame. The filesFrame has three parts: two listboxes that list files on the server and the client in the directories specified by the project name; a set of control buttons to allow a number of transactions take place between the two listboxes: file ‘upload’, ‘download’, ‘release ownership’, ‘lock ownership’, ‘refresh’, etc. These transactions take place once the filename has been highlighted in a specific listbox. By clicking on the file name, the contents of the file are displayed in the text window.

The display of the files in the listboxes can be filtered by clicking the appropriate class selection, e.g. *.bib would display files in this class only. In addition to files, the listboxes also show the current owner of the file. Only the file owner can modify or delete

a file. There are two ways the owner can release ownership of a file: (1) by clicking on the file in the ownership field, or (2) by highlighting the file name and clicking the ‘release’ button. In either case the ownership field will change from ‘userId’ to ‘server’. There are two ways the owner can lock ownership of a file when it is owned by ‘server’: (1) by clicking on the file in the ownership field (it will change from current server to ‘userId’), or (2) by highlighting the file name and clicking the ‘lock’ button. In either case the ownership field will change from ‘server’ to ‘userId’.

editFrame. This frame is invoked by clicking on ‘Text File Viewer/Editor’. The editFrame includes a scrollable text widget in which user can write, modify, or delete text of a file that has been retrieved from the directory on the server or the client – depending on user selection. Besides the text widget, additional widgets in this frame provide functionality such as ‘edit file’, ‘save on server’, ‘save’, and ‘clear’.

stdoutFrame. The stdoutFrame is a scrollable text widget which accepts inputs from standard output and standard error. Its main purpose is to maintain a log of all transactions taking place between the server and the client.

7 Collaborative Experiments

A number of experimental testing of the client/server architecture was taking place for most of the last third of the course. During the last week of the course, there were two sets of student presentations and demos: as a dry run and as a brief presentation/demo during the open house. These demos came in two sets:

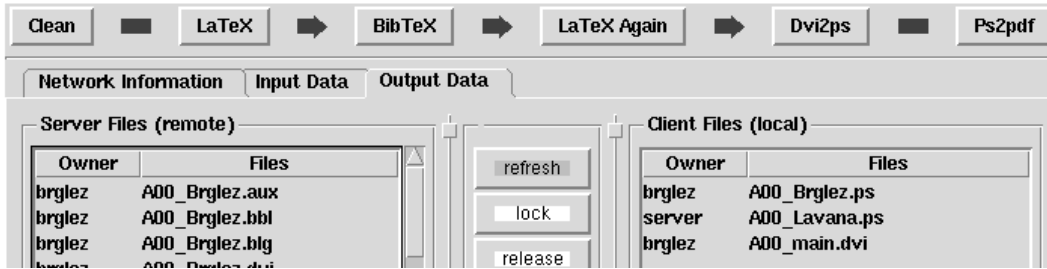
Demo set 1. The complete set of demos that used the universal client described in Figure 5 is summarized in Figure 6:

(a) This flow executes the collaborative document composition as discussed in the preceding sections.

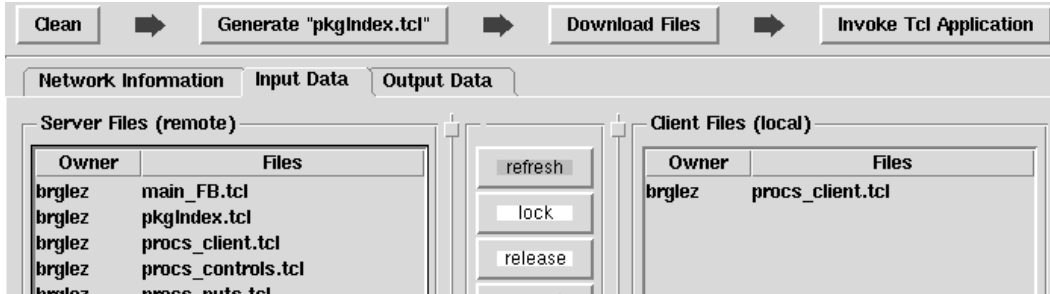
(b) This flow executes the collaborative Tcl/Tk compile. Here, there may be two or more participants working on a complex Tcl/Tk application. Each may be writing and testing a set of procedures in isolation. The question is: will they work together as expected. By uploading the files from several sources to the server, a combined version can be assembled and compiled on the server, then accessed for execution among the participants.

(c) This flow executes the collaborative experimental design. A colleague may have uploaded a

(a) Configuration to execute a collaborative document composition



(b) Configuration to execute a collaborative Tcl/Tk compile



(c) Configuration to execute a collaborative experimental design

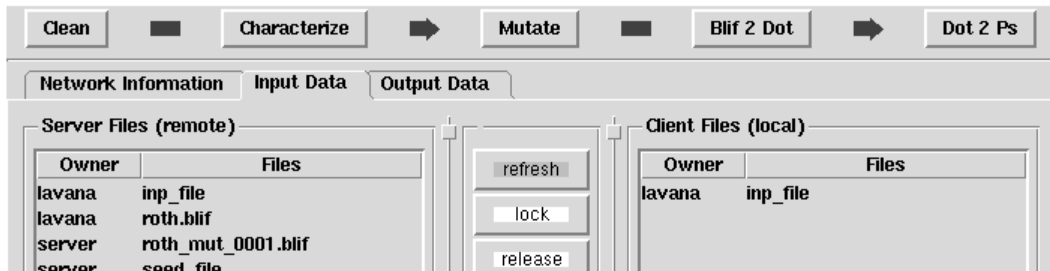


Fig. 6. Three executable configurations of the universal workflow client.

set of reference circuit files to the server to create a mutant class of circuits. These are characterized and laid out as schematics which can be accessed by another team for inspection and further analysis. Such flows are expected to play an important role in the collaborative design of experiments to test the performance of various algorithms [15].

Demo set 2. This demo set consists of the remaining four client implementations devised by student teams. Except for the universal user-reconfiguration of the task flow implemented in Figure 5, all of the client GUI designs have implemented the collaborative task flow for document composition in LaTeX as per original specification in the class. The minor difference in the interface are a reflection of individual preferences and the interpretation of the design specification. The screenshots of these client GUI

designs are available in a technical report [19].

Notably, *all implementations* of the client have tested as executable through a web-browser. The latter included the implementation of the csc591b policy that allowed each participant to download and save files in the directory of the local host – thus overriding the nominal defaults of the web-browser. Currently, such fine-grained security policies are not as readily achievable with browser-based clients written in Java.

Final impressions. As the course concluded, there were really no surprises. The client/server architecture behaved as expected – it allowed multiple students to independently complete the writing assignments about a phase of the project which would then be shared with other participants and included in the overall document such an early draft of this paper. Preliminary experiments, such as rendering

a single-user application collaborative under user-controlled preference also took place – a promising new approach, now described in more detail in the companion paper [1].

8 Conclusions

The material for this paper evolved as part of an experimental course on ‘collaborative computing’. We are no closer to making a definite statement about what actually is the most appropriate definition of ‘collaborative computing’ than we were in the first paragraph of the paper.

In the context of this project, definitions are less important as the expectations we may have of collaborative computing. Bringing together a class of students and learning a scripting language that allows for rapid prototyping of user interfaces and networking concepts, and linking it all to a well-defined project driver, has been an important motivating factor for each participant eager to improve the environment where collaboration can be a rewarding learning experience.

The rewarding experience has been not only to learn the textbook material but also to question existing client/server architecture and to try some new ones.

The projects on the Asynchronous Group Server Architecture and the Synchronizing Group Server Architecture (SGS) along with the respective clients continues as a small project and both the server and the client software is expected to be released for use by the peer community during the year 2000. See

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

for more details.

References

- [1] H. Lavana and F. Brglez. CollabWiseTk: A Toolkit for Rendering Stand-alone Applications Collaborative. In *Seventh Annual Tcl/Tk Conference*. USENIX, February 2000. Also available at <http://www.cbl.ncsu.edu/publications/#2000-TclTk-Lavana>.
- [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] Netmeeting Version 3.0. Published under URL <http://www.microsoft.com/netmeeting>, 1999.
- [4] The Tcl/Tk Consortium. Published under URL <http://www.tclconsortium.org/>, 1998.
- [5] GroupKit Version 5.1. Published under URL <http://www.cpsc.ucalgary.ca/grouplab/groupkit>, 1998.
- [6] R. S. Gray. Agent Tcl: A transportable agent system, 1999. For up-to-date bibliography and software releases, see <http://agent.cs.dartmouth.edu/software/agent2.0/>.
- [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] 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>.
- [9] H. Lavana and F. Brglez. WebWiseTclTk: A Safe-Tcl/Tk-based Toolkit Enhanced for the World Wide Web. In *Sixth Annual Tcl/Tk Conference (Best Student Paper Award)*. USENIX, September 1998. Also available at <http://www.cbl.ncsu.edu/publications/#1998-TclTk-Lavana>.
- [10] M. Harrison and M. McLennan. *Effective Tcl/Tk Programming*. Addison-Wesley, 1998.
- [11] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
- [12] B. B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, 1997.
- [13] TANGO: Collaboratory for the Web. Published under URL <http://trurl.npac.syr.edu/tango>, 1998.
- [14] G. Konduri and A. Chandrakasan. A Framework for Collaborative and Distributed Web-Based Design. In *Proceedings of the 36th Design Automation Conference*, June 1999.
- [15] H. Lavana, F. Brglez, and R. Reese. User-Configurable Experimental Design Flows on the Web: The ISCAS’99 Experiments. In *IEEE 1999 International Symposium on Circuits and Systems – ISCAS’99*, May 1999. A reprint also accessible from <http://www.cbl.ncsu.edu/publications/#1999-ISCAS-Lavana>.
- [16] (La)Tex Navigator, 1999. See <http://www.loria.fr/services/tex/english/index.html>.
- [17] 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>.
- [18] 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-realttime-collaboration.pdf>.
- [19] 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. Technical Report 1999-TR@CBL-01-Brglez, CBL, CS Dept., NCSU, Box 8206, Raleigh, NC 27695, May 1999.
- [20] BWidget Toolkit, 1999. For more information, see <http://www.unifix-online.com/BWidget/>.