Building Real Time Groupware with GroupKit, A Groupware Toolkit

Mark Roseman and Saul Greenberg

Department of Computer Science
University of Calgary
Calgary, Alberta
Canada T2N 1N4
phone: +1 403 220 6087
email: roseman *or* saul @cpsc.ucalgary.ca

Abstract.

This paper presents an overview of GroupKit, a groupware toolkit that lets developers build applications for synchronous and distributed computer-based conferencing. GroupKit was constructed from our belief that programming groupware should be only slightly harder than building functionally similar single-user systems. We have been able to significantly reduce the implementation complexity of groupware through the key features that comprise GroupKit. A *runtime infrastructure* automatically manages the creation, interconnection, and communications of the distributed processes that comprise conference sessions. A set of *groupware programming abstractions* allows developers to control the behaviour of distributed processes, to take action on state changes, and to share relevant data. *Groupware widgets* let interface features of value to conference participants to be easily added to groupware applications. *Session managers*—interfaces that let people create and manage their meetings—are decoupled from groupware applications and are built by developers to accommodate the group's working style. Example GroupKit applications in a variety of domains have been implemented with only modest effort.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*user interface*s; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.4.1 [**Operating Systems**]: Organization and Design—*interactive systems*; H.5.2 [**Information Interfaces and Presentation**] User Interfaces—*user interface management systems*; H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—*synchronous interaction*.

General Terms: Groupware toolkits, computer supported cooperative work.

Additional Key Words and Phrases: User interface toolkits, synchronous groupware, GroupKit.

To be published as:

Roseman, M. and Greenberg, S. (in press) "Building Real Time Groupware with GroupKit, A Groupware Toolkit." ACM Transactions on Computer Human Interaction. Probable year of printing is 1996.

The original submission to ACM TOCHI is available as Technical Report #95/560/12, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada, April 1995. This is a revised version of that paper.

1. Introduction.

Over the last few years, we have been designing groupware for synchronous, distributed conferencing, where two or more distance-separated people work on a shared task in real-time. Our first system, called Share (Greenberg 1990), allowed participants in a distributed meeting to take turns sharing unaltered single-user applications; group interaction was mediated by choosing from a variety of flexible floor control mechanisms to best match the particular style of the meeting (Greenberg 1991). We then built several groupware equivalents of paint and structured drawing programs (Greenberg, Roseman, Webster and Bohnet 1992). GroupSketch was a minimalist bitmapped sketchpad, where all users saw exactly the same things on their display, as well as the multiple pointers of other participants. X/GroupSketch was a second generation version that, amongst its additional features, allowed participants to view different parts of the document. GroupDraw was a prototype of a structured drawing application, where participants could jointly manipulate a variety of objects, such as lines and rectangles, on their display.

Building groupware proved a frustrating experience. Implementing even the simplest systems was a lengthy process, as much time was spent inventing similar ideas over and over again. Some of the common tasks of the programmer include the following items. The set-up and management of distributed processes had to be arranged. Inter-process communication links had to be established. Actions between processes had to be co-ordinated and their internal states updated as people interacted with the system. Similar interface components had to be re-implemented to provide for generic group needs. Session managers had to be supplied so that people could create, monitor, and enter the conferences.

Consequently, we decided to implement a groupware toolkit to support programming of synchronous and distributed computer-based conferencing systems. Our motivation for this project was our belief that:

> A developer using a well-designed toolkit should find it only slightly harder to program usable groupware systems when compared to the effort required to program an equivalent single user system.

We took the tasks common to almost all groupware programming (noted earlier) and transformed them into a set of core user and programmer-centered requirements for a groupware toolkit. These are summarized in Table 1, and are described in more detail elsewhere (Roseman and Greenberg, 1992). The table also lists the rationale behind the requirements. From these requirements, we believed that a toolkit could reduce implementation complexity by providing the following generic features.

- A *runtime infrastructure* would automatically create processes, and manage their interconnections and communications.
- A simple set of *groupware programming abstractions*, built on top of a conventional language and GUI toolkit, would be available to groupware developers. Primitives would include remote procedure calls between application instances, sharing of data, and generation and tracking of conferencing events.
- A set of *groupware widgets* would let developers easily add generic interface constructs of value to conference participants, resulting in better and more usable groupware systems.
- *Session management*, the mechanism by which people create and manage meetings, would be handled separately from the groupware applications. Primitives for constructing different session managers would be available for developers wishing to create custom interfaces that suit the particular needs of a group.

The result of our efforts is GroupKit, a toolkit whose design has been evolving over several years. The first generation of GroupKit, developed in C++ and InterViews (Linton, Vlissides and Calder 1989) has been described in a previous paper (Greenberg & Roseman 1992). Based on our experiences with that system, we constructed a second generation version that has proven to be an even richer platform for developing groupware. Readers familiar with the earlier work should find that the version described here both generalizes and extends the earlier system.

GroupKit and its applications still run on Unix workstations under an X11 environment. However, the system now uses the interpreted Tcl language and Tk interface toolkit (Ousterhout 1994) and the Tcl-DP socket extensions (Smith, Row and Yen 1993). GroupKit developers build their applications using Tcl/Tk as well as the extensions provided by our toolkit. GroupKit and the underlying systems are all freely available via anonymous ftp; details are provided at the end of this paper[1].

[1]All the systems that comprise GroupKit are under active development, with new versions of software appearing periodically. This paper is mostly based upon GroupKit 3.1, Tcl 7.4, Tk 4.0, and Tcl-DP 3.2.

Design Requirement	Rationale	Examples
User-centered		
Support multi-user actions and awareness of others over a shared visual work surface.	Many groupware applications can be seen as shared work surfaces, and studies (e.g. Tang, 1991; Gutwin and Greenberg 1995; Dourish and Bellotti, 1992) have attempted to characterize some of the properties necessary to support collaborative work. Supporting these in a toolkit can encourage developers to build more usable applications.	• telepointers • graphical annotations • multi-user scrollbars • gestalt views
Provide support for structuring group processes, but do so in a flexible enough fashion to accommodate the diverse needs of different groups.	Rather than adopt a single model for how groups should interact, a toolkit should provide a range of facilities to support the needs and working styles of different groups, while allowing application developers to extend these to support specific needs (Greenberg, 1991). A toolkit should support building applications relying on either social protocols or highly structured and automated process models.	Different policies for: • floor control • session management • access to conferences • treatment of latecomers
Integrate groupware with conventional ways of doing work.	Groupware should not pose a barrier to "individual" ways of doing work, but instead should be smoothly integrated (Ishii and Kobayashi, 1992). Access to both single-user applications or even non-computer resources (Ishii, 1990) is important, as is the availability of traditional communication mediums.	• shared terminals • telephone links • video conferencing
Programmer-centered		
Provide technical support to deal with multiple distributed processes.	Groupware systems are composed of multiple processes communicating over a network, and toolkits can augment the operating system level support by simplifying the creation, interconnection, and teardown of these processes. A toolkit can also provide communications models and concurrency control mechanisms that abstract away from the operating system.	• session management • process creation • locate other processes • multicast abstractions • session persistence • message serialization and data locking
Provide support for shared data	Distributed groupware should be able to share its common data easily, should be able to detect and take action when data is changed. Concurrency control should be available to keep data consistent (Greenberg and Marwood 1994).	• shared environments • data serialization and locking • binding callbacks to environment events
Provide support for shared data and an extensible shared graphics model.	As many groupware applications can be seen as shared visual work surfaces, a toolkit should support the creation and manipulation of both generic and application specific objects on a graphical work surface. Paradigms such as Abstraction-Link-View (Patterson, 1991) can be supported by the toolkit to facilitate this.	• shared graphics primitives • object concurrency control • separate view from object representation

Table 1. Core requirements for a groupware toolkit

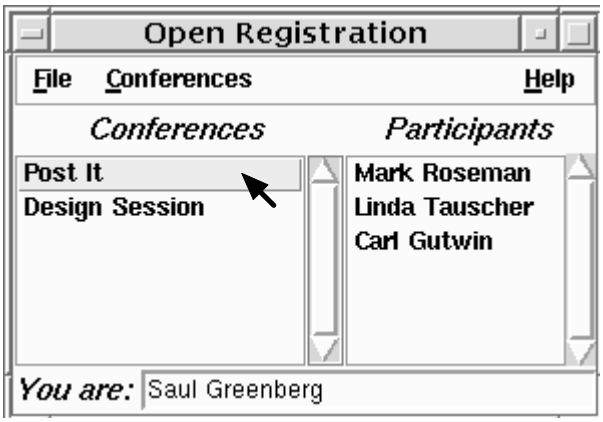

Figure 1. The Open Registration session manager.
Two conference sessions are shown, with three participants present in the "Post It" conference.

The paper begins with an overview of GroupKit. It shows how an end user sees systems constructed with the toolkit, and what a GroupKit program looks like. Subsequent sections detail GroupKit's features—its runtime infrastructure, its groupware programming abstractions, the set of groupware widgets, and its session management. These sections show how some of these features work in practice by including both screen snapshots and code fragments from existing applications. The examples also illustrate the wide variety of systems that can be built in GroupKit. The paper then evaluates GroupKit by examining the effort required to build groupware applications in it. It closes by comparing GroupKit to other groupware toolkits. A video is also available (Greenberg and Roseman 1994) that captures the dynamics of many of the screen snapshots described in this paper.

2. Overview

Before delving into technical details, it is worth getting an overall feel for GroupKit. To set the scene, this section begins by showing what end-users of programs built in GroupKit may see. It then takes the developer's view, by tracing through a simple GroupKit program.

The scenarios presume that users' computers are running X Windows within a Unix environment, that computers are interconnected using TCP/IP protocol over the Internet, and that the GroupKit software has been installed. Users may be located anywhere on the Internet as long as their network connection does not suffer excessive latency (which would compromise interactive performance of some applications). Because GroupKit can run in most Unix systems, the actual machine type does not matter[2].

2.1. An Example of an End-User's View of GroupKit

This section walks through an example GroupKit session, where we will see a user monitoring conferences in progress, joining two existing conferences, and then creating a new conference. While the scenario illustrates actual systems provided in the GroupKit release, it is important to remember that these are just examples of systems that programmers could build with the toolkit.

First, the user (Saul) invokes a *session manage*r, in this case the "Open Registration" manager (Figure 1). In the "Conferences" pane, Saul sees that two conferences are in progress: "PostIt" and "Design Session". By selecting one of them, he can then see who is in a particular conference (the list in the "Participants" pane).

Next, Saul joins the "PostIt" conference by double clicking its name, which adds him to the list of participants. The PostIt Editor then appears on his display (Figure 2, left window). With this simple GroupKit application, he can type a short message and send it to one or more participants. The selected participants will see the message appear in a pop up

[2]For example, GroupKit has been successfully installed on Sun, HP, RS6000, and Silicon Graphics workstations as well as PCs running Linux.

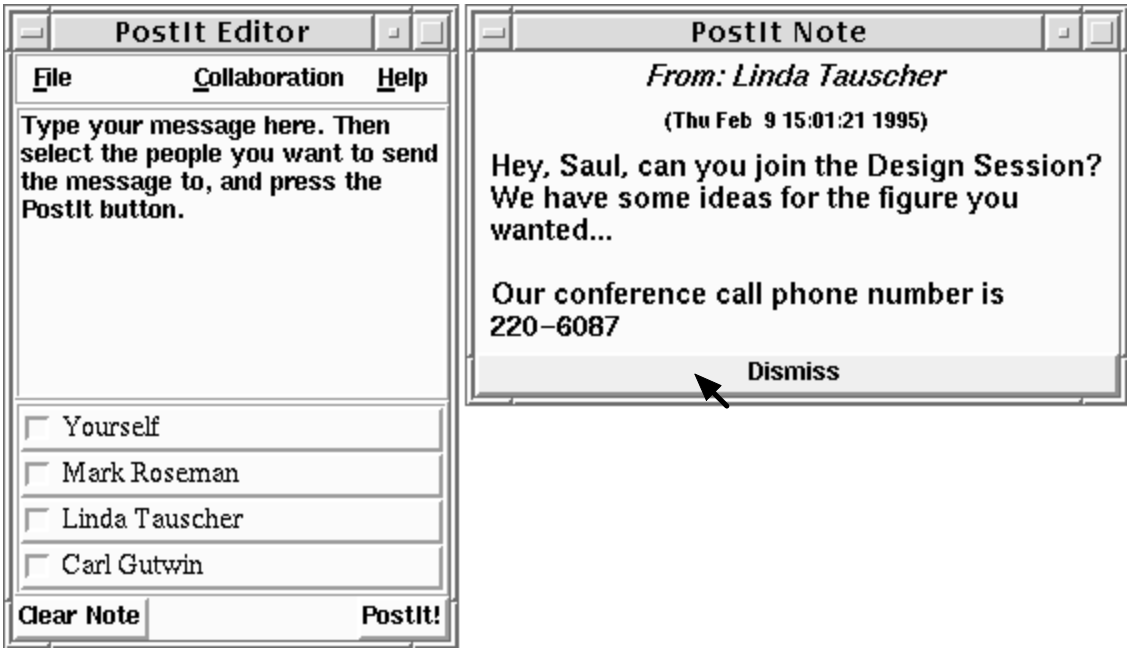

Figure 2. The PostIt application.
Notes are created through the PostIt Editor, and received as a PostIt Note.

Figure 3. The GroupSketch multi-user sketching application,
showing the shared drawing, the local cursor, and two remote telepointer widgets.

window. In Saul's work community, everyone uses PostIt to see who is available, and to invite people into conferences. Shortly after joining, Saul receives a PostIt message from Linda inviting him to join the "Design Session" conference (Figure 2, right window).

Saul joins the "Design Session" conference via the session manager and GroupSketch, a multi-user sketchpad that allows simultaneous drawing, then appears on his display (Figure 3). It contains the group drawing being worked on, as well as the telepointers of the other participants. The voice connection is made through a telephone conference call. After discussing the figure, Saul leaves the conference by selecting the "Quit" option from the "File" menu.

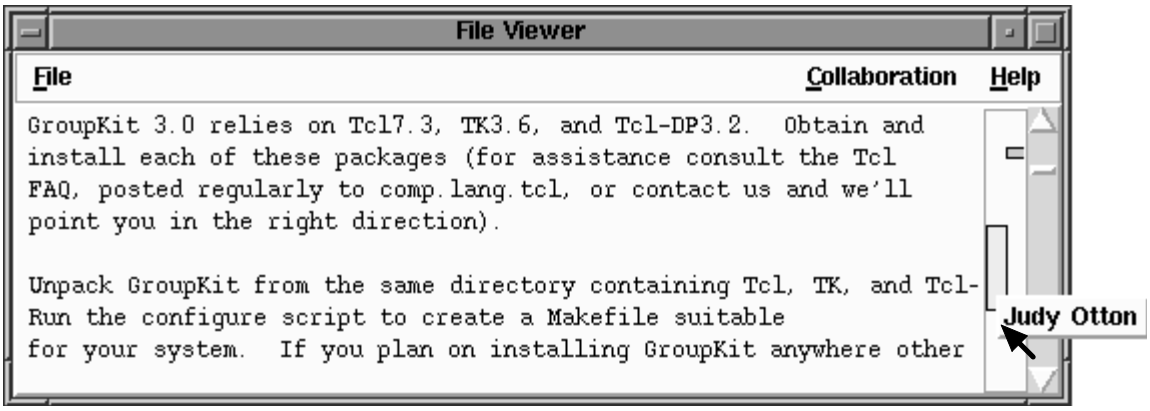

Figure 4. The FileViewer relaxed WYSIWIS application, showing the multi-user scrollbar widget

A bit later on, Saul receives a phone call from his colleague Judy, who wishes to discuss a document. Saul decides to create a new conference that runs the FileViewer. This application provides a relaxed what-you-see-is-what-I-see (WYSIWIS) view of a text document[3], and displays multi-user scroll bars showing what each person is able to see. From the "Conferences" pull-down menu in the session manager (Figure 1), he selects "File Viewer" from the many applications listed, which adds its name to the Conferences pane in every session manager. FileViewer appears on his display, and he then selects a file to view in it. Judy joins the conference, and they continue their discussion around the shared view (Figure 4).

This scenario illustrates several important points about GroupKit.

- GroupKit supports real-time distributed multi-point conferences between many users.
- GroupKit systems include both *session managers* for managing conferences, such as the Open Registration system in Figure 1, and *conference applications* which are the actual groupware tools, as shown by the PostIt, GroupSketch, and FileViewer applications in Figures 2, 3 and 4.
- Groupware designers can build session managers that match the working styles of their audience and organization. The "Open Registration" system illustrated here is one that presents a free-for-all policy on session management; other examples are presented later in this paper.
- GroupKit is a generic toolkit, in that a variety of conference applications relying on either text or graphics can be built. Other example applications will be illustrated in later sections.
- GroupKit is not a media space system. While we strongly believe that voice communication is necessary for most real-time interaction, GroupKit does not directly support audio or video conferencing. For now, we expect users to use other systems, such as telephones. Optionally, an application programmer could have GroupKit automatically invoke an external computer-based audio/video system as a GroupKit conference whenever a conference connection is made.[4]

2.2. A Developer's View of GroupKit: An Example Program

The above section illustrated what an end user of GroupKit may see. This section presents an application developer's point of view, by walking through a complete groupware application called "hello world". Even though it is a simple example (in keeping with its genre), it is far from trivial. Not only does it demonstrate several important GroupKit constructs, it is probably the only "hello world" program that actually says hello to the world!

Every participant in a "hello world" conference runs their own copy of the program. Each person sees a window on their screen containing a button labelled "Hello World" (Figure 5, top). When any user presses the button, the buttons on all participants' screens are changed to display briefly a greeting from the user who pressed the button. For example, if Mark Roseman pressed the button, all participants would see "Mark Roseman says hello!" (Figure 5, bottom).

[3]In strict WYSIWIS systems, all users see exactly the same thing on their displays. In relaxed-WYSIWIS, users may have different viewports into the shared workspace and/or different representation of their contents (Stefik et al, 1987).

[4]GroupKit could be extended to provide direct multimedia communications. We decided against this because: a) we did not have the resources to duplicate existing work in media spaces; b) it would make GroupKit both platform and hardware dependent; c) GroupKit's ability to invoke existing on-line audio/video systems appeared to be a reasonable compromise; and d) the telephone company already supplies an excellent audio channel, which suffices for many situations.

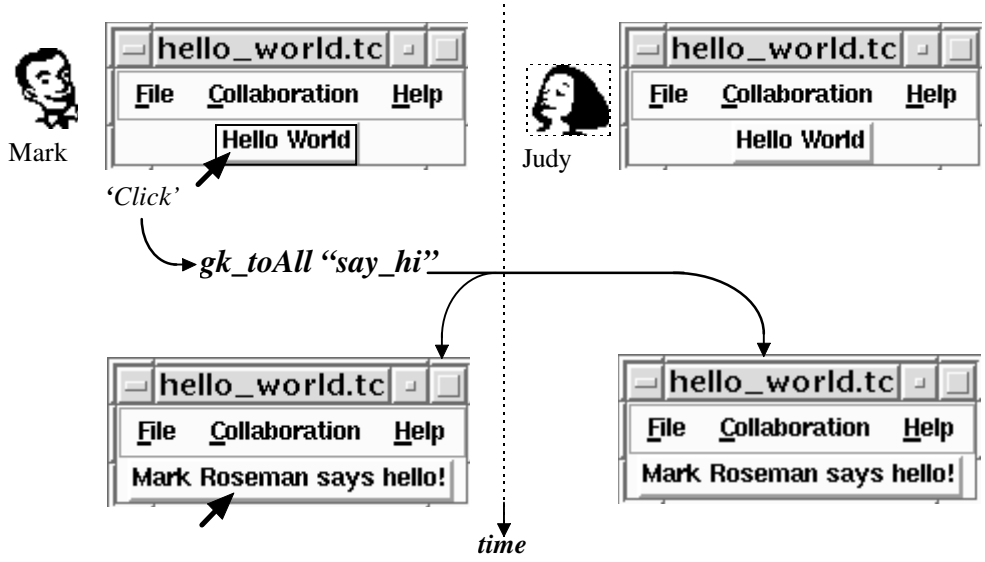

Figure 5. The Hello World application running on two displays.
The sequence of events shows the procedure 'say_hi' being multicast to all replicated processes.

To create this GroupKit conference, the application programmer would need to write a program similar to that shown below. Most of the code is standard Tcl/Tk concerned with providing the user interface; only the parts in bold signify the GroupKit code needed to make the program group-aware.

```
1   gk_initConf $argv
2   gk_defaultMenu .menubar
3   pack .menubar -side top -fill x
4   set greetings "[users local.username] says hello!"
5   button .hello -text "Hello World" \
                  -command "gk_toAll say_hi \"$greetings\""
6   pack .hello -side top
7   proc say_hi {new_label} {
8       .hello configure -text $new_label
9       after 2000 {.hello configure -text "Hello World"}
10  }
```

Line 1 initializes the application, and asks GroupKit's runtime architecture to automatically manage such things as maintaining socket connections between this local process and the copies being run on the other participants' workstations.

Line 2 creates GroupKit's default menu bar, a widget that provides access to some generic groupware facilities (other widgets are discussed in Section 5). It gives the participant the ability to leave the conference gracefully (a "Quit" option in the File menu), lets them see what other conference participants are present (a "Participants" option in the Collaboration menu), and provides access to help topics (the Help menu). Line 3 calls the standard Tk function that places the widget at the top of the window.

In line 4, we create a string (called `greetings`) that will serve as a message from the local participant e.g., "Mark Roseman says hello!". This line queries a special data structure provided by GroupKit called an *environment* that maintains information about local and remote users. In this case, the code within the square brackets asks the environment to return the name of the local user, which is used to construct the string. Environments have other features, and are discussed further in Section 4.3.

Lines 5–6 create a standard Tk button, initially giving it the label "Hello World". The GroupKit function `gk_toAll` is attached to it as a callback, and is executed when the button is pressed by the user. This procedure is a *multicast remote procedure call* (Section 4.1) that arranges for other functions (in this case the procedure `say_hi`) to be executed not

only on the workstation where the button was pushed, but on the workstation of every user in the session. This sequence is illustrated in Figure 5.

Finally, lines 7–10 contain the `say_hi` procedure. Line 8 changes the button's label to contain the procedure's argument (the greetings), and line 9 changes it back to "Hello World" after 2 seconds. When `gk_toAll` multicasts the procedure execution, each person's display will show the same greetings message, as illustrated in Figure 5.

If we wanted to make this example slightly more sophisticated, we could add the following lines to the end of the program. This will automatically change the button's text whenever a new user enters into the conference, e.g., "Saul Greenberg just arrived!" would be displayed.

```
1–10 as above
11  gk_bind newUserArrived {
12      set new_user_name [users remote.%U.username]
13      say_hi "$new_user_name just arrived!"
14  }
```

These lines illustrate how developers can take advantage of *events* that are automatically generated by GroupKit. Here, the developer asks that some code be automatically executed whenever a new user enters the conference, as indicated by the `newUserArrived` event. In line 11, GroupKit's `gk_bind` command attaches the code on the next two lines as a callback to the `newUserArrived` event. When this event is triggered by the arrival of a new user, the code is executed locally. Line 12 will retrieve the name of the arriving user; in this case `%U` indicates which user has arrived, and the environment query `[users remote.%U.username]` provides the name of that user[5]. Line 13 calls the say_hi procedure, whose argument is now the arriving user's name concatenated onto the string "just arrived!". The interaction between events and environments are discussed further in Sections 4.2 and 4.3.

Finally, the program is made available to the session manager by editing a configuration file. All that has to be specified here is a default conference name (e.g., Hello World) and the location of the program.

This program, simple as it is, illustrates several important points about developing groupware in GroupKit.

- GroupKit's *conference applications* are decoupled from *session management*. The developer does not have to worry about creating new processes or managing communication connections.
- Developers have access to information about the conference through GroupKit's *environments* data structure.
- Developers know that conference applications are executed as communicating replicated processes, one for each participant.
- Developers can synchronize GroupKit conferences through *multicast remote procedure calls*.
- GroupKit provides a suite of special *groupware widgets* which are easily attached to a groupware program. These widgets encapsulate tools useful to end-users.
- Developers can attach callbacks to *groupware events* that allow them to note and take action when, for example, people enter and leave conferences.

As the emboldened parts of the program above suggests, doing the groupware hello world program *is* only slightly harder than its equivalent single-user version.

3. The Run-Time Infrastructure

One of the design requirements listed in Table 1 said that a groupware toolkit should provide technical support for programmers dealing with multiple distributed processes. We believe this is best satisfied by including a run-time infrastructure that actively manages these processes. Its responsibilities would include: process creation, location, interconnection, and teardown; communications set-up such as socket handling and multicasting; and groupware-specific features such as providing the infrastructure for session management and persistent conference sessions (Section 6). This section presents the actual run-time infrastructure supported by GroupKit. It describes the different types of processes GroupKit creates and maintains, and the interactions between them. Later sections will introduce the programming abstractions available to developers that are supported by this infrastructure.

[5]We designed the syntax of `gk_bind` and its use of a `%<field>` specification for retrieving information about events to be similar to Tk's event binding mechanism, making it familiar to Tcl/Tk programmers.

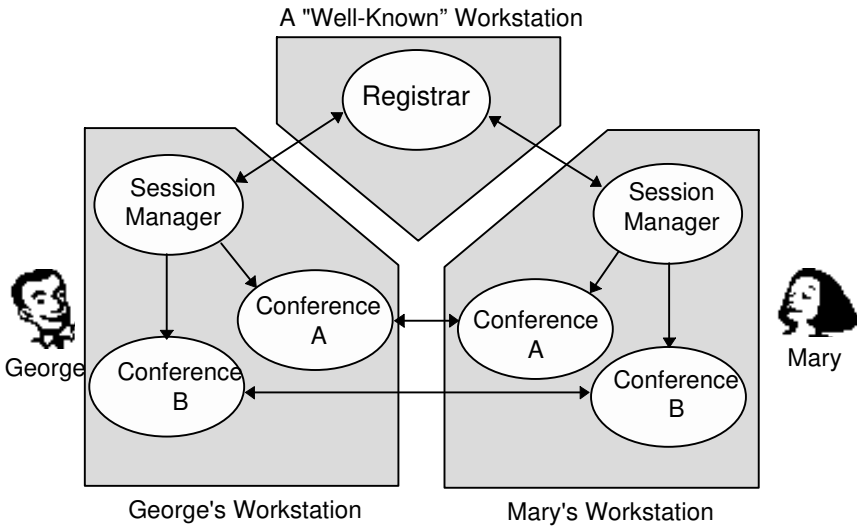

Figure 6. An example of GroupKit's runtime process model

GroupKit's run-time infrastructure consists of a variety of distributed processes arranged across a number of machines. Figure 6 illustrates an example of the processes running when two people are communicating to each other through two conferences 'A' and 'B'. The three large boxes represent three workstations, the ovals are instances of processes running on each machine, and the directed lines joining them indicate communication paths. Three types of GroupKit processes are shown: a *registrar*, *session managers*, and *conference applications*.

Registrar. The Registrar is the first process created in a GroupKit session. There is usually one Registrar for a community of conference users, and its address is "well-known" in that other processes know how to reach it. This is the only centralized process required by GroupKit's runtime infrastructure.

When session manager processes are created, they connect to the Registrar. The Registrar maintains a list of all conferences and the users in each conference. It thus serves as an initial contact point to locate existing conference processes. The Registrar itself does not have a policy on how conferences are created or deleted, nor on how users join or leave conferences. Rather, it is the session manager which directs the Registrar to add or delete conferences and users; if requested, the Registrar then relays these changes to other session managers.

Session Manager. The session manager is a replicated process, one per participant, that lets people create, delete, monitor, join, or leave conferences. The session manager provides both a user interface and a policy dictating how conferences are created or deleted, how users are to enter and leave conferences, and how conference status is presented. We have already seen the Open Registration manager in Figure 1; other quite different managers can be created by the developer to suit different registration needs (see Section 6). Internally, session managers do not communicate with each other directly, but discover changes in conference state as information is propagated through the central list maintained by the Registrar.

Conference Applications. A conference application is a GroupKit program, and is invoked by the user through the session manager. This program, created by the application developer, is a groupware tool like a shared editor, whiteboard, game, and so on. Conference applications typically work together as replicated processes, in that a copy of the program runs on each participant's workstation[6]. We call a process instance a *conference process,* while the set of conference processes working together is called a *conference session*. Different types of conference sessions may be running at the same time, all managed by the user's one session manager.

[6]While the conference processes that make up a conference session usually execute the same program, they can differ substantially as long as they are well-behaved. That is, the messages, data, and remote procedure calls sent between the differing programs must be understandable and must produce a correct response (the same is true for replicas of the session manager). As an example, Section 4.3 will illustrate how two different versions of the "hello world" program can co-exist.

While the applications do not have to worry about low level details of session management, they may wish to be notified via an event when some high-level session change does occur. They may also retrieve information about conference participants, which is maintained automatically by GroupKit. This was shown in the "hello world" example, where events tracked the arrival of a new user, and environments were used to retrieve both the local user's and an arriving user's name.

Figure 6 illustrates these concepts. The centralized Registrar is running on its own workstation, and its address is known to this particular GroupKit community. Two users, George and Mary, have each invoked their own session managers, such as the Open Registrar shown in Figure 1, which communicate to the Registrar. Mary has used her session manager to create conference session A, which spawns the actual conference A process (e.g., the GroupSketch program). To track session events, her session manager and conference process will maintain an internal communication link to each other. George could then join that conference through his session manager, which creates a replicated copy of the conference A process; a communication path between the two copies is automatically established. The other conference B (e.g., FileViewer) is established in a similar fashion. Both Mary and George would each see three windows: the Open Registrar session manager, GroupSketch (conference A) , and FileViewer (conference B).

This infrastructure is maintained entirely by GroupKit. The conference application code does not need to take any explicit action in process creation or communication set-up. Instead, as mentioned in Section 2.2, the application may just ask to be notified through an event when particular session activities occur. The conference processes that comprise a conference session can also co-ordinate with each other through the high-level programming abstractions provided by GroupKit, as discussed in the next section.

4. GroupKit's Programming Abstractions

The programmer-centered requirements in Table 1 state that a groupware toolkit should provide support for co-ordinating multiple distributed processes and for sharing data between them. In GroupKit, common chores are handled by the run-time infrastructure. However, application-specific demands are left in the hands of the developer to manage through GroupKit's programming abstractions. Developers are aware that they are writing distributed groupware programs, and it is their responsibility to make all conference processes in a conference session behave correctly, to take action on state changes, and to share relevant data. A failure to do so can result in different processes having different state information, leading to inconsistencies and possible system failure. Developers can also use these abstractions to code in appropriate levels of support for structuring a group's work process, which satisfies one of the user-centered requirements in Table 1. GroupKit provides the developer with three programming abstractions to support these tasks: *multicast remote procedure calls*, *events* and *environments*, as described below.

4.1. Multicast Remote Procedure Calls

One style of programming GroupKit applications relies heavily on a specialized form of remote procedure calls (RPCs) to communicate, share information, and trigger program execution between replicated application processes in a session. Internally, GroupKit uses Tcl-DP's RPC mechanism to send a Tcl command from the local process to a remote one, and to execute it there as if it were called locally (Smith, Row and Yen 1993). For the groupware programmer, GroupKit abstracts these RPCs even further; first by not requiring programmers to track the addresses or even the existence of other application processes; and second by automatically multicasting procedures to some or all conference processes. [7]

The several forms of GroupKit RPC calls that are available to the programmer differ in who the messages are sent to. The `gk_toAll` procedure, seen in the "hello world" example, multicasts a procedure to all conference processes in the conference session, including the local user. This results in the same procedure being executed everywhere, which was illustrated in Figure 5. The `gk_toOthers` procedure multicasts a command execution to all the remote conference processes in the session except the local process that generated the call. Finally, `gk_toUserNum` sends the command to the conference process of a particular user in the session. In this case the "address" is the unique user number of a conference participant maintained internally by GroupKit and easily retrieved by the programmer.

[7] This is more than a cosmetic add-on to Tcl-DP. The conference's run-time infrastructure interacts with the session managers to locate conference processes of other users. It automatically creates and destroys the socket connections as users join and leave, and hides details such as the file descriptors attached to sockets. Instead, a GroupKit programmer is presented with a unique id number for each user, which is used to find information about that user and to relay requests to their processes.

Figure 7. The Brainstormer application.
Ideas are entered in the bottom and added when “enter” is pressed.

The multicast RPC paradigm is an effective way of turning a single user application into a multi-user application. Imagine a brainstorming conference, where users anonymously enter ideas by typing them into a text entry widget and pressing return. The ideas are then appended to a shared list seen by all users (Figure 7). First consider the single user case. A standard Tcl/Tk procedure like the following might be defined that inserts a new idea into our list:

```
1   proc insertIdea idea {
2       .idealist insert end $idea
3   }
```

When executed, this would insert the idea (passed as an argument and held in the `idea` variable) into the Tk text widget called `.idealist`. The multi-user version would just change line 2 to:

```
1   proc insertIdea idea {
2       gk_toAll ".idealist insert end $idea"
3   }
```

The difference here is that not only do we insert the idea into our own list, but we direct every other conference process to insert the idea into their own lists.

The other GroupKit RPCs give the programmer different capabilities and somewhat more flexibility. For instance, `gk_toOthers` lets a programmer take different actions locally and remotely. Consider the example above. The programmer may wish to display local ideas (the ones entered by the local participant) in blue, and all remote ones (the ones entered by all other participants) in red. Assuming that a procedure `insertColouredIdea` has already been defined that adds ideas to the list in an indicated colour, we could substitute line 2 in the original example for:

```
1   proc insertIdea idea {
2a      insertColouredIdea blue $idea                    #locally executed
2b      gk_toOthers "insertColouredIdea red $idea" #remotely executed
3   }
```

While simple, the multicast RPC model provides a powerful yet flexible approach to distributed programming. The programmer does not have to know the addresses of other conference processes or track process creation and destruction as people enter and leave the session; the calls work the same way whether one user or twenty users are in the conference session.

In the brainstorming conference example above, it is possible (although in practise quite rare) for ideas to be inserted into the lists in different orders. This is because the GroupKit RPC calls described so far do not guarantee that all sites will receive messages in the same order. This is fine for applications where messages are independent of each other or when the effects of out of order events are inconsequential (Greenberg and Marwood 1994). When order is important,

Event	Description
newUserArrived	a new participant has arrived in the conference session
userDeleted	a participant has just left the conference session
updateEntrant	a late entrant into the conference session needs updating

Table 2. Conference application events automatically generated by GroupKit.

there is an additional `gk_serialize` RPC that works like `gk_toAll`, except that messages are serialized through a central process (one of the conference processes), thus guaranteeing in-order delivery.

Finally, most of GroupKit's RPCs are non-blocking. Once the request for remote procedure invocation is made, the program continues its execution without waiting for a reply from remote processes. We have taken this approach because we noticed that GroupKit programmers rarely require its RPCs to return a value. This ensures that conference processes are not delayed or blocked in the event of network latency or crashes on remote machines. However, we have provided a blocking variant of `gk_toUserNum` that waits for a value to be returned for the few times it is needed.

4.2. Events

Another abstraction in GroupKit is the *event,* which provides a way for conference applications to be notified when various things happen. These can be events automatically generated by GroupKit, such as session events that occur when participants join and leave the conference session, or custom events created by an application developer. If an application wants to take action on a specific event type, a callback can be created which is automatically executed when the actual event is generated. This section describes both the event mechanism and the types of events generated by GroupKit. As will be seen, events are a critical component of several schemes GroupKit uses to satisfy some of the requirements listed in Table 1 e.g., flexible handling of latecomers, synchronizing distributed processes, and noticing changes to shared data.

An event consists of an event type and a set of attribute/value pairs that provide information about the event. The set of attributes of an event depend on its type. Programmers trap particular types of events through an event handler or *binding*. Bindings are specified using the `gk_bind` command, and take the form `gk_bind [event-type] [action]`. The action contains callback code that is executed when the event occurs. The event's attributes can be accessed within this code using *percent substitutions*, which are modeled after Tk's event mechanism. A simple example of its use was already shown in lines 11-14 of the "hello world" conference, and a more complex example will be provided shortly.

GroupKit's run-time infrastructure automatically sends three different event types to conference processes, as described in Table 2. The first two event types are generated when users join and leave the session, as a conference process may want to take special action when this happens. One simple example was already shown in the hello world program, where arriving users caused a new message to be displayed in the button. As another illustration, consider TextChat, a multi-point "talk" application. Figure 8 shows its interface, where each conference participant has a tiled window on the display. When a participant types, the text is added into their particular window. For the conference session to behave correctly, each conference process must track the arrival of a new participant to the conference, and create a tiled chat window to contain their text. In line 1 of the code fragment below, the programmer has created a binding via `gk_bind` that gets called when a `newUserArrived` event is automatically generated by an arriving user. When this event happens, the code on line 2 is executed. Here, the identity of the arriving user is retrieved via `%U` (the user number of the arriving user is one of the attributes of this event), and the programmer-defined callback `makeChatWindow` is invoked. When the callback is executed (line 3) the window is created.

```
1   gk_bind newUserArrived {
2       makeChatWindow %U
3   }
4   proc makeChatWindow usernum { ... }            # Create the chat window
```

Similarly, each conference process would check for departing participants and attach a callback that deletes their particular window from the local display.

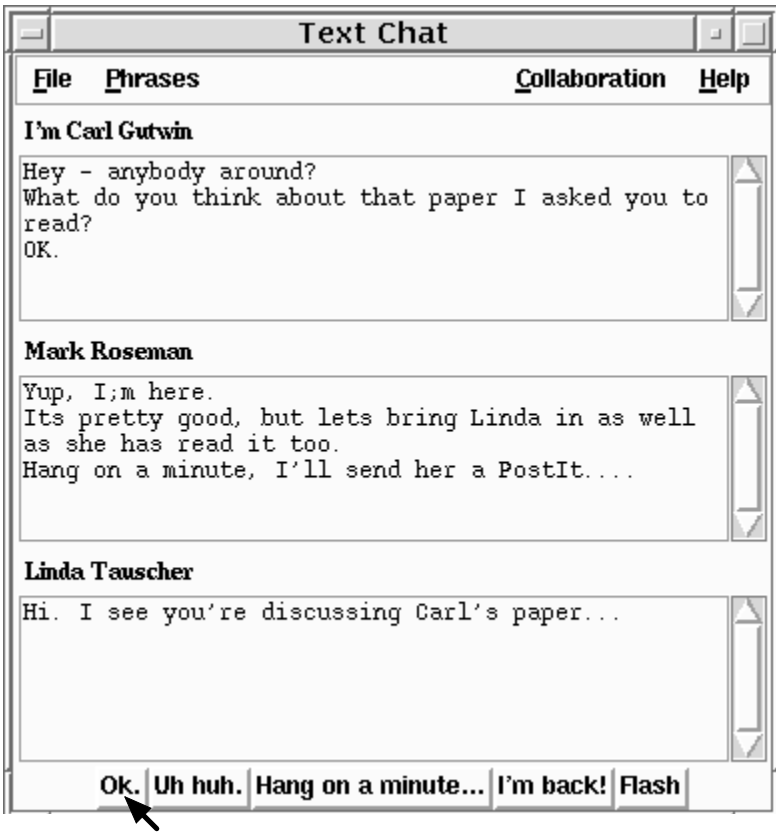

Figure 8. The Text Chat application.
Windows show the text typed by conference participants. The local user's text goes in the top window.

Programmer's can attach callbacks to them if desired.The third event in Table 2 is used to handle latecomers to conferences that are already in progress. By monitoring the arrival of the latecomer, their conference process can be brought up to date by one of the other conference processes in the session, usually by sending it the existing state of the conference. This event is automatically generated when the latecomer joins the conference. Unlike the other events, GroupKit's run time infrastructure sends it to only one of the conference processes in the session, because only one should update the latecomer[8]. For example, the GroupSketch program that was illustrated in Figure 3 attaches a binding to this event that, when executed, will cause a single GroupSketch process to send the entire drawing to the latecomer as a display list. The TextChat program would use it to send the text contents of all the existing chat windows, and the Brainstorming program would pass on the set of ideas generated so far. The Hello World program did not need to do anything, as all its replicas automatically return to the same basic state.

There is no generic way for a groupware toolkit to handle latecomers. Perhaps the conference just needs to bring the latecomer up to date, as in the examples above. Even so, the optimal strategy may depend on the type of application. A complete transcript of all user activities since the start of the session may be sent, or a snapshot of the current screen, or an abstract data model that can be used by the newcomer's conference process to generate the display. Alternatively, the application could be a game where the latecomer may be assigned a particular role that differs from the other players. As handling latecomers is application-specific, GroupKit leaves the decision of how to handle the newcomer up to the programmer of the conference application.

Finally, application developers can generate their own custom events. This can be useful in more complex applications, where a change being handled in one part of the program can generate an event to notify other parts of the program (or

[8]GroupKit's mechanism for determining which process to send the event to is rather arbitrary: it is the one identified by the lowest user number.

other processes) of the change. The end of Section 4.3 will illustrate how events can be used to propagate changes of program state across applications.

4.3. Environments

While multicast RPCs and events can be used to co-ordinate conference replicas, GroupKit applications can also communicate via *environments*. Environments are a dictionary-style data structure containing *keys* and associated *values*. Instances of environments running within different conference processes can communicate with each other. The net result is that environments can provide a shared data structure (a requirement in Table 1): changes to an environment in one conference process are propagated to the environment instances of the other processes. This section describes how environments work, and provides an example application demonstrating the programming paradigm they support.

As a data structure, programmers store and retrieve related information within an environment via a hierarchical key. For example, GroupKit tracks information on all users in the session via the `users` environment, introduced in the previous code fragments. `[users local]` returns a record about the local user, while `[users remote]` is a record containing information on all other participants. A record for a particular participant is retrieved by appending their unique user number, and actual values are retrieved by refining the hierarchical key even further e.g., `[users remote.3.username]` returns the name of remote participant #3.

GroupKit's environments are more than a data structure. First, they can act as active values. The programmer can bind callbacks—similar to GroupKit events—to an environment so as to receive notification when a new piece of information is added to it, when information is changed, or when information is removed. Second, environments can be shared so that a change in one conference process' environment is propagated to the other processes in the conference session. By combining sharing and event generation, a change in one process' environment can change the data in others. Because the change generates events, the corresponding actions are triggered at all sites.

GroupKit's environments are an effective way to support the programming model advocated in part by Smalltalk's model-view-controller (Krasner and Pope 1988) and later by Patterson. Both argue for separating the view of an object from its data representation. In the Rendezvous toolkit, Patterson (1991) encourages developers to create groupware applications using its abstraction-link-view (ALV) model, whose constructs are:

- a shared underlying data abstraction,
- a view of the abstracted entity that may differ for each user,
- a constraint (called a link) that automatically adjusts the view when the data abstraction is changed.

The power of this idea is that quite different views can be generated from the same data abstraction. In GroupKit, environments provide the underlying abstraction; when changes to this environment are made—by local or remote users—events are generated. These events are monitored by the interface code, which adjusts the view to reflect the state of the environment. This effectively gives GroupKit part of its extensible and flexible shared graphics model, helping it satisfy another of the requirements listed in Table 1.

To see how this works in practice, here is a new version of hello world where changes begin after line 3 of the original program. The abstraction used is simply a variable that contains a greetings message. First, line 4 creates the `message` environment and line 5 adds the key `greetings` (initialized to the value "Hello World") to the environment. The `-bind` option causes the environment to generate events whenever it is altered, while the `-share` option propagates these changes to the corresponding environment in the other conference processes. Next, the button is defined as before on lines 6 and 7, except its callback just changes the contents of the `greetings` key. Line 9 attaches a binding to the message environment (the "link" that is activated when its data changes)[9]. Lines 10-11, executed when the event is triggered, adjusts the view to match the abstraction by changing the button's label.

```
1   gk_initConf $argv
2   gk_defaultMenu .menubar
3   pack .menubar -side top -fill x
4   gk_newenv -bind -share message
5   message greetings "Hello World"
6   button .hello -text "Hello World" \
```

[9]The event pattern `changeEnvInfo` is generated whenever an environment's existing attributes are changed. Other event patterns detect the addition of new attributes, or deletions of old ones.

```
7     -command {message greetings "[users local.username] says hello!"}
8   pack .hello -side top
9   message bind changeEnvInfo
10          .hello configure -text [message greetings]
11          after 2000 {.hello configure -text "Hello World"}
12  }
```

Different programs in a single conference session can now have different views on the same data abstraction. The version below of hello world monitors what other participants are doing by creating a time-stamped transcript of all the greetings messages being passed around. It starts after line 5 of the version above. Lines 6 and 7 create a standard Tk listbox which will serve as a transcript. Line 8 creates the same binding, but now the code in lines 9-10 sets the view by first getting the time (via a Unix command) and the new greeting, and then by inserting them at the beginning of the transcript.

```
1-5 as above
6   listbox .transcript
7   pack .transcript
8   message bind changeEnvInfo {
9           set text [concat [exec date +%H:%M:%S] ":" [message greetings]]
10          .transcript insert 0 $text
11  }
```

One final concern with environments is concurrency control. As we have argued elsewhere (Greenberg and Marwood 1994), we believe concurrency control needs are highly application dependent, and that no one mechanism would suffice. We are now adding concurrency control into GroupKit's environments. The default is "no concurrency." Although this means that environment replicas can get out of step with each other, there are quite a few groupware situations where this matters little in practise, where the social protocol of the group can be used to minimize conflicts, or when the performance penalty is not worth the cost of undermining an application's responsiveness (Greenberg and Marwood 1994). As an option, programmers can now serialize all changes to an environment, either by directing changes through a single conference process or through a separate, dedicated process. It is even possible for more ambitious programmers to extend environments with their own concurrency control policies; the mechanisms to do so are described elsewhere (Roseman, 1995). As our own work on environments progresses, we will add a variety of other concurrency control policies, such as conservative and optimistic locking.

Using this programming paradigm, we have developed a simple structured graphics editor, initially as a single user system. Its environment contained the abstraction about the objects being edited. A user's interaction resulted in changes to this environment, and changes were propagated back to the view via the event mechanism. To convert this program to a multi-user application, we just had to add two lines of code initializing GroupKit, and the flags to the environment declaration specifying it was to be shared with other users in the session. The point here is that group-aware events and environments can, in some cases, almost eliminate the coding differences between single and multi-user systems.

5. Groupware Widgets

Along with the above programming abstractions, GroupKit provides developers with a collection of multi-user *groupware widgets* that helps satisfy many of the user-centered design requirements in Table 1. Using GroupKit's widgets, the developer can easily add groupware features to applications that conference participants will find valuable. GroupKit even has a rudimentary "class builder", a widget kit that developers can use to create their own widgets, or to extend Tk and GroupKit widgets.

Some researchers have created multi-user analogues of conventional single-user widgets, such as buttons, menus, and simple text editors, and investigated how to make the sharing of widgets flexible enough to fit different work situations. Interested readers are referred to Dewan's work on Suite (1991), as well as work by Smith and Rodden (1993). While their approach is important, our research interests rest in widgets that support activities found only in group work. We are currently working on three classes of widget functionality: *participant status, telepointers,* and *location awareness*.

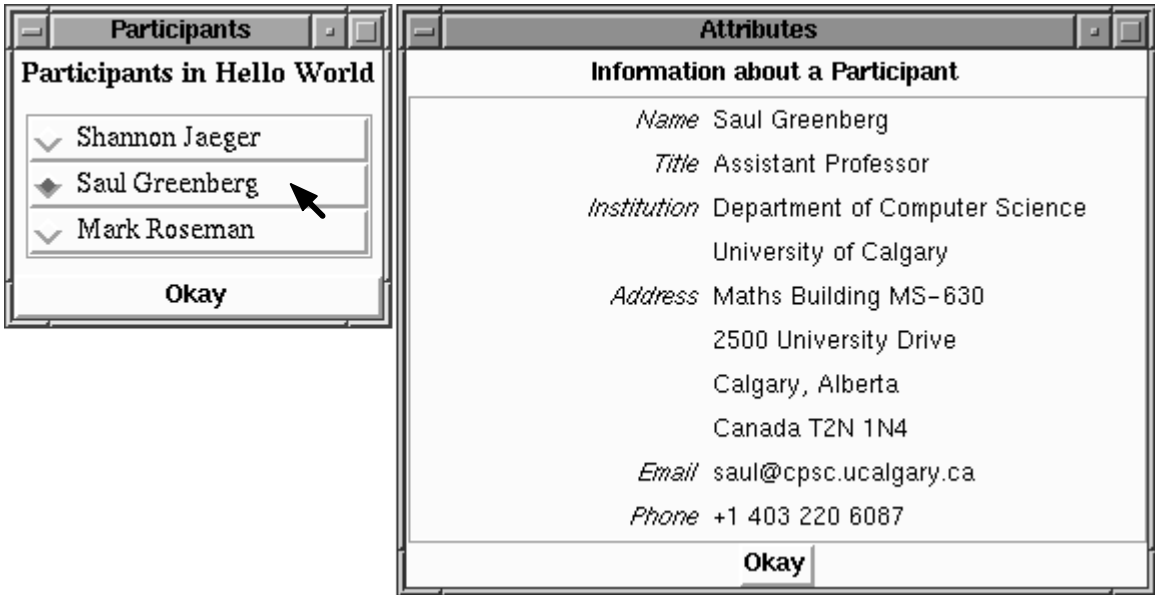

Figure 9. The Participants widget.

5.1. Participant Status

As people enter and leave a conference, other participants should be able to see their comings and goings, much in the same way that we can see people arrive into a room. Because these people may be strangers, it would be handy to find out some information about them.

GroupKit's participants widget, illustrated in Figure 9, partially provides this information. It lists all participants in the current conference session (left side). The list is dynamic, and will automatically update as people enter and leave. When a participant is selected, information about them is displayed (the pop-up window on the right). This could include contact information (as shown), a picture of the person, and any other material that person wished to pass on about themselves. While this widget is easily added to any GroupKit application, we also make it available to all end users through a menu selection in GroupKit's menu bar widget.

We are now including within the participant widget (and GroupKit in general) the ability to monitor the activity of participants. This includes whether they are actively using their computer (and thus are more likely to be available for real-time conversation) as well as the ability to see how willing others are to receive calls. The rationale behind these features and a few non-GroupKit prototypes that we have built are discussed in Cockburn and Greenberg (1993).

5.2. Telepointers

Studies of small face to face groups working together over a shared work surface reveal that *gesturing* comprises about 35% of the group's activities (Tang 1991). Gestures are a rich communication mechanism. Through them, participants indicate relations between the artifacts on the display, draw attention to particular artifacts, show intentions about what they are about to do, suggest emotional reactions, and so on. In video-based groupware, gestures are transmitted by actually showing hand movements over the work surface (Tang and Minneman 1990; Ishii 1990). Computational groupware systems typically use telepointers (also known as multiple cursors) to provide a simpler but reasonably effective mechanism for communicating gestures (Hayne, Pendergast and Greenberg 1994). Unfortunately, modern window systems are tied to the notion of a single cursor, and application developers must go to great lengths (and suffer performance penalties) to implement multiple cursors. They are usually implemented as graphical objects embedded within the application (Greenberg and Bohnet 1991), as widgets or windows placed on top of the shared view, or as "overlay layers" (Beaudouin-Lafon & Karsenty 1992; Roseman & Greenberg 1992) where the telepointers are part of a transparent graphical layer that sits above the shared view layer.

The InterViews version of GroupKit implemented telepointers through overlays, while the current Tcl/Tk version uses bitmap widgets that are placed on top of particular widgets in the shared view. Regardless of the implementation

technique used, GroupKit's telepointers can be created and attached to the underlying widgets with two lines of code. For example, Figure 3 shows the telepointers in a GroupSketch session. The local user's cursor is the normal one provided by the window system, while the two remote users are (in this case) shown by the small circles. The code in GroupSketch that adds the telepointers to the Tk canvas widget containing the shared drawing surface is:

```
1   gk_initializeTelepointers
2   gk_specializeWidgetTreeTelepointer .canvas
```

GroupKit's telepointers can partially handle relaxed-WYSIWIS displays. Instead of tying a telepointer to a window, a programmer can attach it to particular widgets and their children (this is the purpose of line 2). The telepointer is always drawn relative to the widget, rather than the application window. For example, we have built a tic-tac-toe game, motivated by the one built in the Rendezvous toolkit (Hill et al, 1994). Its relaxed WYSIWIS view allows a participant to reposition their game board in their application window without affecting the views of other participants. Even though the board may be in the upper right corner of one participant's view and the lower left of another's, the cursor will be drawn in the correct position over the board.

We are now working on the idea of *semantic cursors* that are tied to the semantics of what a person is doing rather than their relative location in a shared view. This is important for relaxed WYSIWIS displays, where people have different views of a document. View difference could be a matter of spatial orientation (the location of items across displays differ relative to each other, as with the tic-tac-toe example above), in representation (one person sees data as a graph, the other as a table), and in detail. The idea is to show a person's focus of attention in the same semantic location on either display, perhaps with information being attached to the cursor itself to indicate what the other is doing. Although quite preliminary, we believe semantic cursors will be a convenient way for participants to be aware of what others are doing in relaxed WYSIWIS situations. For example, we have prototyped a relaxed WYSIWIS popup menu that uses semantic cursors to show others what menu items are being chosen. To the person using it, it behaves as a conventional menu; a mouse-click over a menu button causes the popup to appear on the display, and a menu item is chosen from the list by selecting it. However, the other participants only see the cursor move to and remain on top of the menu button—the popup is not shown. Instead, a small label is attached to the cursor that shows only the name of the particular menu item being moused over and/or selected. Other examples are shown in Gutwin, Stark and Greenberg (1995).

5.3. Location Awareness

One type of relaxed-WYSIWIS in groupware allows participants to have different viewports into the larger shared work surface. This is natural for real work sessions, for people often do individual work that they later bring back to the group. In real life, we are kept aware of what others are doing, sometimes by speech, sometimes by seeing what others are working on through our peripheral vision. This helps us co-ordinate our work. Because these cues may not be available in the groupware channel, we include two *location awareness* widgets in GroupKit that tell a participant where other people are working in the shared work-surface: *multi-user scrollbars*, and *gestalt viewers*.

The *multi-user scrollbar*, which was illustrated on the right side of Figure 4, is inspired by the one provided by the SASSE text editor (Baecker, Nastos, Posner and Mawby 1993). It supports collaborator awareness by indicating the relative locations of users within a large document. The right half of the scrollbar is a normal single-user scrollbar, allowing the user to move within the document. To its left is a vertical bar showing the relative location of each conference user, identified by a unique colour. The name of the bar's owner can be displayed by mousing over the bar. The bar's position and size is continuously updated as participants scroll through the document or change their window size. Unlike SASSE's scrollbars which are built for a particular application, GroupKit's multi-user scrollbars are generalized widgets that can be attached to any scrollable object. The work required by the programmer is virtually the same as using conventional Tk scrollbars, as they can be attached through a single line of code.

The *gestalt viewer*, also inspired by SASSE, is similar in spirit to the multi-user scrollbars, but is much richer in function. It works by presenting a miniature of the document overlaid by coloured boxes showing the actual viewport of each participant in the session (Figure 10). These boxes are active interface objects, as the users can scroll to a new location by grabbing and moving their box. The miniature provides structural cues about the document as well as participants' locations, which could help a user better understand where their collaborators are and what they are working on.

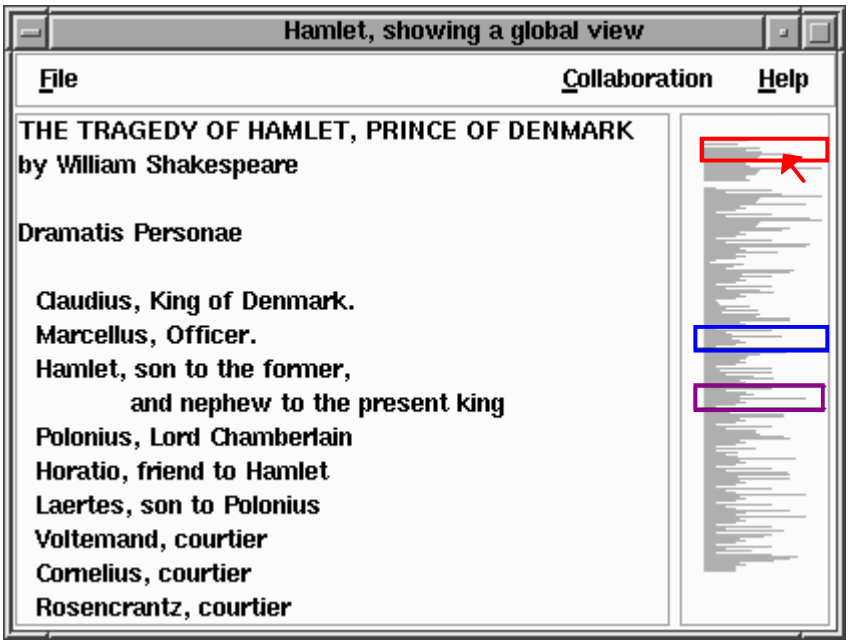

Figure 10. The Gestalt Viewer widget.

The document on the left is shown in miniature on the right. Boxes represent each participant's viewport.We are currently experimenting with other location awareness widgets. We are trying to generalize the notion of the gestalt viewer to a variety of document types. We have also created a "what you see is what I do" scheme that shows a full-scale view of the area immediately surrounding another person's cursor (usually a subset of their view). We are also interested in awareness of changes on a shared view: one of our prototypes lets people replay the significant stops in a person's interaction over the view, while another prototype highlights any changes that have occurred since the participant last attended the view. Details are provided in Gutwin and Greenberg (1995).

6. Session Management

Not only do GroupKit developers build conferences, but they can construct session management interfaces as well. This is in sharp contrast to most toolkits that force a single (often rudimentary) session management interface onto its applications. As mentioned in Table 1, a groupware toolkit should provide support for structuring group processes in a flexible enough fashion to accommodate the diverse needs of different groups. To show how this requirement was applied to session management, and to show the variety of session management policies that can be built, this section describes several different policies and their interfaces that have been constructed in GroupKit. It also outlines briefly how managers can be constructed.

A session manager typically controls and presents an interface to the following tasks:

- creating new conferences,
- naming conferences,
- deleting conferences,
- locating existing conferences,
- finding out who is in a conference,
- joining people to conferences,
- allowing people to leave them, and
- deciding whether conferences persist when all users exit.

For example, the interface of the session manager could present these as explicit steps that a user takes to begin and maintain the collaboration. These could also be implicit actions, where (say) the act of jointly editing an artifact automatically initiates the collaboration (Edwards 1994).

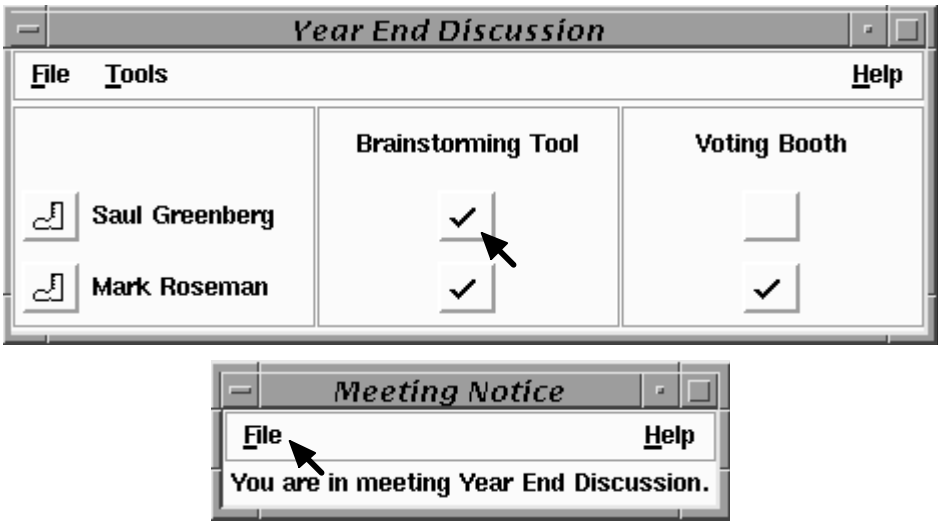

Figure 11. The Centrally Facilitated session manager. The Facilitator's control panel for managing sessions is shown on top. Participants just receive a notice that they are in the meeting.

Being able to provide different interfaces for session management is an important aspect of supporting the working patterns of a group. We strongly believe that one of the obstacles to groupware use is the difficulty of starting up a groupware session (Cockburn and Greenberg 1993). The obstacle may be in terms of usability (e.g., the system is difficult to initiate) or social (e.g., the policy the system imposes is not acceptable to the group). Session management must be more than an afterthought added to the applications, and should be tuned to the needs and collaboration patterns of the target user group. This is why GroupKit actively supports the development of session management systems by developers.

6.1. Example Session Managers

To illustrate the possibilities, this section briefly describes the policy and interface of three session managers constructed in GroupKit. The first, using an open registration model, is well suited to informal collaborations amongst equal status participants. The second example follows a facilitator model, where a facilitator manages the selection of tools for the users of, say, a group decision support system. The last example uses a rooms metaphor, and is more suited to sets of collaborative tasks that are intermixed over longer periods of time.

Open Registration. This manager has been described in previous sections (Figure 1). Its interface supports a permissive policy of creating and joining conferences, which seems well-suited to collaborations between equal status participants. It has two other features not mentioned previously. First, a person can invite other people into session via a menu option. Second, a conference and its state can, upon request, persist after the last participant leaves it; if new people arrive or old ones re-enter, the conference resumes where it left off. Alternatively, conferences can be automatically deleted when the last participant leaves it. Persistent sessions are handled as part of GroupKit's core, and can be added to any session manager (the ones below include it as well).

Centrally Facilitated. This pair of session managers is designed for a structured meeting controlled by a single person, the facilitator. Individual users do not initiate or join conferences. Instead, the facilitator, through his or her session manager, controls the session by inviting participants into the session, by selecting appropriate tools for them (the groupware applications), and by activating the specific tools that a participant should see on their display. Figure 11 shows the facilitator's session manager (the upper window). The names of meeting participants are in the left column. By toggling the appropriate button in the table to their right, the facilitator can selectively add or remove the tool to a participant's display. The participant's session manager, shown in the bottom window, is much simpler. It informs participants that they are in a particular meeting, and lets them leave the conference. No other action is allowed.

Rooms-Based Session Management. The last example, illustrated in Figure 12, combines an open policy with a rooms-based metaphor. Users can create virtual meeting rooms, and stock them with meeting tools through the pulldown menu options. Each room lists who is in them, what tools are running there, and shows the degree of privacy desired (via the door icon). If no one is in a room, the tools remain available as they are treated as persistent conference sessions. People can freely move between rooms. When they enter a room they are joined to all the conferencing tools located in the

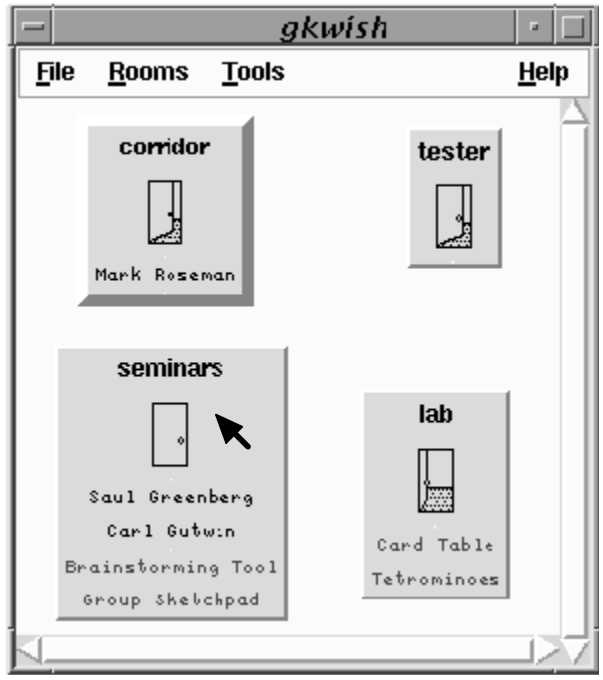

Figure 12. The Rooms session manager. Each room displays the users and applications in it, and the door state provides a cue on how welcome others would be into its room.

room; when they leave the room, any tools used in the room are left behind. This system could serve the needs of collaborators working on many tasks over a period of time, allowing them to easily move between tasks. It also serves as a meeting place, where people can see who is around in what room, and converse with them after entering the room. An example of its use is illustrated in Figure 12 after four rooms have been created. The seminar room has two people in it, running two applications; their closed door indicates that they do not wish to be disturbed[10]. The lab room has two games going, although no one is currently in it. The tester room, which is completely empty, is a convenient place for people to place and test software being developed. The corridor is a special room that users enter when they first start the session manager.

We are now experimenting with other session managers. These include a call-based system similar to the way people place telephone calls, and a document based manager where groupware connections are triggered automatically as people view and edit common documents.

6.2. Implementation of Session Managers

Building session managers in GroupKit is similar in style to building conferences. Session managers communicate with the central Registrar—and thereby with each other—to manipulate lists of active conferences and their users. GroupKit stores this information in environments, and changes to these environments generate events. Developers respond to events by providing their own event handlers or bindings, or by using GroupKit's default handlers.

The event-based approach allows developers to define their own session management policy, governing the actions taken when certain things happen. For example, when a local user tries to create a conference, GroupKit automatically generates an event indicating that a "user requests a new conference." The open registration policy discussed above responds by actually creating the conference and joining that user to it. By trapping events, event handlers can also ensure that the session manager's user interface component is kept up to date when interesting things happen.

GroupKit's session management facilities are based on *open protocols,* a technique we have developed for building flexible groupware (Roseman and Greenberg, 1993). Briefly, a central server (the Registrar) provides a data structure (replicated via the environments), but specifies no policy for how the data structure is to be used. Clients to the Registrar (the session managers) specify the policy by the selection of operations they perform. Maximum flexibility is achieved by providing open access to the Registrar's data structure via a protocol or interface of small but powerful operations (e.g., add or delete conference). Clients may be different, as long as they are well behaved with respect to each other and to the policy. For example, the two session managers used in the centrally facilitated example are programmed to work well together. In contrast, if a group mixed session managers and one person created a conference using the open registration system when others were using the centrally facilitated system (which was not designed to interact with it), the conference name would not appear on the facilitators list and therefore would be inaccessible.

Table 3 enumerates the handful of different session management events that GroupKit will generate. The column on the right lists some of the default event handlers GroupKit provides to deal with these events. Different session managers will use these in different ways to create their policy. The open registration system, for example, relies entirely on these handlers to implement its policy. Only a small percentage of its 175 lines (about 15 lines) is concerned with the policy, while the bulk of the system builds the user interface for that policy, and tracks what events should be displayed to the

[10]The version illustrated presents the doors as a social cue, as the software does not stop people from entering even if the door is closed. This is a matter of personal philosophy, as we believe that people are good at mediating their own social situations. If desired, the software could be programmed to enforce the closed door policy by disallowing new entrants.

Event	Description	GroupKit-provided Handler
foundNewConf	A new conference has been created	Add conference to local environment and generate a newConfApproved event
newConfApproved	Approval of a new conference	None (can use to update interface)
foundDeletedConf	An existing conference has been removed	Remove conference from local environment and generate a deleteConfApproved event
deleteConfApproved	Approval of a conference deletion	None (can use to update interface)
foundNewUser	A new user has been added to a conference	Add user to the local environment and generate a newUserApproved event
newUserApproved	Approval of a new user	None (can use to update interface)
foundDeletedUser	A user has left a conference	Remove user from local environment and generate a deleteUserApproved event
deleteUserApproved	Approval of a user deletion	None (can use to update interface)
lastUserLeftConf	The last user in a conference has left	None (session manager may choose to delete conference)
conferenceDied	A conference process we created has terminated	Tell the registrar to delete the local user from the conference
userRequestNewConf	User asked for a new conference	Create the requested conference

Table 3. Session Management events and handlers.

user (largely via monitoring "approved" events). The complexity of the facilitated manager is comparable to the open registration system. In contrast, the rooms-based manager is more complex. While it builds upon the event handler structure, its developer had to add extra data structures to maintain information on the rooms into its policy section.

7. Discussion

We began this paper by claiming that a developer using a well-designed toolkit should find it only slightly harder to program groupware than equivalent single user systems. To see how GroupKit fared in this test, we took the groupware applications that we had built, and measured their complexity in terms of lines of code[11]. Their names, length, and brief descriptions are tabulated in Table 4. Comments, blank lines, and help documentation (used by GroupKit's on-line help widget) were removed from the count, leaving only the functional code.

[11]Added complexity could have been measured by comparing GroupKit programs with single-user equivalents written in Tcl/Tk. However, such a measure proves misleading. First, most of our programs are examples that show off groupware aspects, and do not have the extensive application features that typically require many lines of conventional code. Since we believe that the amount of groupware-specific code will not increase linearly with conventional code, the size difference would be grossly exaggerated. Second, there are no sensible single user equivalents of some of the programs we have developed. Third, if some of the programs were written with the proper separation of view from abstraction, there would be little difference between the two (although this would require GroupKit's environments, they could be programmed in an non-sharing mode, essentially a single user extension to Tcl/Tk).

Name of Application	Lines of Code	Description
Hello world	10	Pressing a button changes its contents on all screens to display the person who pressed it (Figure 5).
Simple Sketchpad	30	A minimalist multi-user sketchpad that lets people sketch simultaneously; includes telepointers.
GroupSketch	311	A stroke-based sketchpad that includes: telepointers drawing, erasing, moving of lines, setting of line colours and thickness, and saving / restoring of images to files. (Figure 3)
FileViewer	51	A relaxed WYSIWIS file viewer, with multi-user scrollbars, plus the ability to load files into view (Figure 4)
Minimalist Brainstorm	19	A relaxed WYSIWIS brainstorming facility for anonymous entry of ideas into a list.
Brainstormer	82	An enhanced version of the tool above, with multi-user scrollbars and the ability to save and load ideas to files (Figure 7).
PostIt	126	A facility for creating Post It notes, for sending them to selected participants in a conference, and for receiving notes (Figure 2).
TextChat	140	A multi-point text-based chat facility. The version measured is slightly simpler than the one shown in Figure 8, as it excludes the ability to make custom text phrases.
Tic Tac Toe	152	A replica of the Tic Tac Toe system built in the Rendezvous toolkit (Hill et al 1994). Players are assigned the roles of X, O or observer, whose semantics are enforced by the system; the board can be on different locations on different screens, and visual feedback of a player's movement is provided.
Hyper	213	An object-based drawing package that lets people create circles and link them with arrows. Arrow links are maintained by constraints when circles are repositioned.

Table 4. The length of sample groupware applications built in GroupKit.

What is striking from the table is that all of the programs, even those with reasonable functionality, are remarkably short, with the largest being around 300 lines long. As well, most took only a few hours to a few days to build. While programming brevity is partly due to the simplicity of building interfaces in the Tcl/Tk language, the point is that the groupware component of these programs did not increase their length unreasonably. Another telling point is that the original GroupSketch and X/GroupSketch systems, built without a groupware toolkit, each took about 3 months to program and were about 5000 lines long. The GroupKit version is only 300 lines, and took a few days to code. Complexity decreased by more than an order of magnitude.

We also talked to application programmers—ourselves, others in our group, and external people who had downloaded the software to their site. Most found GroupKit relatively easy to learn, understand, and program. When stumbling blocks did occur, they tended to be in the "single user" part of the application code. For example, the developer of the Tetrominos game spent far more effort making Tk's canvas objects flip and rotate, than making them group-aware. As another example, a group in Germany had created a multi-user text editor with turn-taking; about two-fifths of this 500 line program involved creating single-user controls for setting the local font and text colour of the editor.

Application programmers also found GroupKit's programming abstractions straightforward. GroupKit widgets are incorporated into the interface in the same programming style of single user widgets, and multicast RPCs are a natural

extension of the way normal callbacks are used. In fact, several single-user programs written in Tk were transformed into partially group-aware versions merely by prefixing their callbacks with `gk_toAll`. For example, TkSol, a thousand-line single user solitaire game, was converted to a partially group-aware version by a student at the University of Toronto. He modified 12 lines to include the `gk_toAll` RPC, added 1 line to initialize GroupKit, and added 2 lines for telepointers.

Another person had developed a single-user microworld called TurboTurtle, an educational simulation environment used by learners to explore Newtonian physics. A student experimented with the world by changing its properties (gravity, friction, etc.) and seeing how this changes the movement of a ball (the turtle). We challenged him to make TurboTurtle group aware (Cockburn and Greenberg 1995a,b). He writes of his experiences:

> "I was honestly stunned by the simplicity of making TurboTurtle group-aware with GroupKit. Prior to actual coding, I was familiar with GroupKit's design infrastructure, and I was concerned that working with it would require a multi-layered approach to group-awareness: demanding that some events be communicated through the Registrar, some through the conference application, and so on. As it turned out, my (the programmer's) interface to group-awareness facilities was highly "task-oriented" and therefore natural. For example, I had a procedure named `do_bounce` in the single-user application; to get everyone's replicated application to "do a bounce" I simply appended `gk_toAll` before the procedure name. All the problems of handling the communications are happily entirely abstracted away from the programmer. My experiences in adding dynamic updates of the system state for latecomers to a TurboTurtle session were equally positive. Similarly so for relaxed WYSIWIS components of the system For example, I wanted to let one user view the turtle's speed as a vector of speed and direction, while another to view it as the x and y components of velocity: this particular facility required *no* change to the original code, because, by default, local changes are not propagated to other users."

Mixed in with his praise is some criticism; some of the widgets (especially telepointers) were somewhat fragile, and the groupware version of TurboTurtle performed slower than the single user version. However, this does not detract from the value of the programming abstractions used. He continues...

> "For empirical evidence on the ease of using GroupKit, here's a log of my experience with modifying the original 1000 lines (or there about) of TurboTurtle. Most group-awareness facilities were added to the system in 2 days of scattered attention (probably a total of 3 hours work). The remaining facilities were added over another 2 days, adding approximately 3 hours more. There are now a total of 80 lines that contain some element of GroupKit. Around 50 more lines were necessary to separate accessing a value (due to the local user moving a slider, say), from the transmission of the value using `gk_toAll`. The cognitive difficulty of reading the code is probably higher, but not significantly so."

Other people have commented favourably on GroupKit as a rapid prototyping environment. In their evaluation of groupware toolkits, Urnes and Nejabi (1994) say of GroupKit that "the combination of an interpreted language together with good session management provided an invaluable tool for quick testing of ideas in a multi-user setting." This is true in our own experiences as well. The scripting language of Tcl/Tk together with the rapid creation of groupware sessions keeps turnaround time for modifying and testing code quite short[12].

While these are encouraging signs, programming with GroupKit's abstractions does not eliminate all groupware complexity. The programmer must consider the interaction between the processes that are being co-ordinated by multicast RPCs, events and shared environments; unconsidered side effects can cause the unexpected to happen. There is also a craft to using GroupKit constructs effectively. Multicast RPCs usually demand that the programmer consider what local actions should be taken and what variables should be set before the procedure and arguments are multicast. Similarly, events and environments are at their best when conference applications are written in a style similar to Smalltalk's model-view-controller (e.g., Krasner and Pope 1988), or Rendezvous' abstraction-link-view model (Patterson 1991). If problems do occur, debugging can be hard because the interaction between conference processes can be non-deterministic and difficult to envisage. As well, programmers now have the additional burden of updating latecomers. While this has been relatively straight forward to do in many GroupKit applications, it does require extra

[12]Urnes and Nejabi (1994) evaluated an earlier version of GroupKit. On the downside, they noted that building relaxed WYSIWIS systems in GroupKit was non-trivial. However, the version tested provided only multicast RPCs; we believe our recent addition of environments as a means to separate the view from the data significantly eases this programming task.

effort. Finally session managers, unlike applications, require a fairly sophisticated knowledge of GroupKit's architectural underpinnings and are thus harder to code.

Another problem is that GroupKit applications may suffer a performance penalty if there is a high degree of parallel activity by users.

1. In addition to responding to local user events and executing the appropriate callbacks, the processor now has to broadcast these events to other participants. It also has to handle incoming events from remote participants as well, which triggers even more local callbacks,.
2. Latency and bandwidth restrictions in communication links can delay message communication; even though the local actions may be handled quickly, the delay can make the entire system seem slower.
3. Because operating systems and programming languages are not designed with groupware in mind, some operations can be overly expensive. For example, GroupKit now implements its telepointers as a Tk label widget placed on top of other Tk widgets, which is an expensive graphics operation. This would be vastly faster if the X window system contained a feature that could handle multiple cursors at the kernel level. As another example, Tk's programming model requires us to update all idle tasks in order to force processing of communication packets. Because this includes (perhaps unnecessary) screen updating, this too becomes an overly expensive operation.

However, none of the above problems detracts from the versatility of the programming abstractions, and we believe that future incarnations of GroupKit or similar toolkits combined with an operating system tuned to groupware demands could minimize these performance penalties.

As a final point, the ideas in GroupKit should be portable to other toolkits. Indeed, many of its essential components were first realized in GroupKit 1.0 (Roseman and Greenberg, 1992), which was implemented in the C++ based InterViews toolkit (Linton, Vlissides and Calder 1989). It was later rewritten in Tcl/Tk with only modest effort. More recently, a member of our team built a Smalltalk prototype that shared some of GroupKit's features. Of course, the difficulty of rewriting GroupKit will depend upon the language. Tcl/Tk, for example, has the advantages of an easy to program scripting language and high-level widget set, while Tcl-DP eliminates much of the tedious coding of sockets. Still, other languages could provide programming advantages that Tcl/Tk lacks, such as support for threads, decent name-space management, and an object-oriented widget set with true inheritance capabilities.

In summary, the ideas in GroupKit work. While it is slightly harder to program groupware applications than single user applications, the extra effort required is quite reasonable.

8. Related Work

This section will relate GroupKit to a few of the ideas found in other groupware toolkits. We warn that direct comparison between groupware toolkits can be misleading, as many were built to address quite different problems or were designed to emphasise different aspects of groupware construction.

We begin by considering toolkits that differ substantially from GroupKit. First, many groupware toolkits address disparate application domains. For example, the ConversationBuilder (Kaplan, Tolone, Bogia and Bignoli, 1992) and Strudel (Shepherd, Mayer and Kuchinsky 1990) are used for constructing speech act protocols. Oval is used to build semi-structured messaging and information management systems (Malone, Lai and Fry 1992). Lotus Notes, although not a programming toolkit, lets people develop and tailor a wide variety of asynchronous applications (Lotus Inc.). Second, even toolkits within the domain of real-time interaction handle different niche problems. Shen and Dewan's (1992) Suite toolkit applies only to highly structured text objects and investigates how flexible access control mechanisms are incorporated into them. Knister and Prakesh's (1990) DistEdit provides groupware primitives that could be added to existing single user text editors to make them group aware. DistView, produced by the same group, is oriented towards a mostly strict-WYSIWIS approach to sharing window components and underlying data via an object replication scheme (Prakesh and Shim 1994). Smith and Rodden's (1993) SOL considers design features for making single user widgets shareable.

Examples of toolkits that do overlap somewhat with GroupKit are Rendezvous (Hill et al, 1994), Weasel (Graham and Urnes 1992), ObjectWorld (Tou et al 1994), Share-Kit (Jahn 1994),and Conference Toolkit (Bonfiglio et al 1989). While it is beyond the scope of this paper to review these toolkits, GroupKit's relation to some of their more prevalent ideas are described below.

Multicast RPCs. Several systems use multicast RPCs (discussed in Section 4.1) as its sole programming abstraction. Share-Kit uses C and the Unix RPC mechanism to build its multicast layer, and its programmers must register a procedure and its argument formats as an RPC and use special keywords to invoke them. In contrast, GroupKit lets any procedure be executed as an RPC, which greatly minimizes housekeeping. The Conference Toolkit uses a routing table to let developers specify the routing of data between application instances. The equivalent control over routing is provided in GroupKit through its `gk_toOthers` and `gk_toUserNum` calls. Unlike these toolkits, GroupKit provides other programming abstractions as well (e.g., events and environments), and programmers can intermix these styles according to application demands and personal preferences.

Separating the view from the abstraction. Section 4.3 has already discussed how different views of an object can be generated from the same data representation. In Rendezvous, programmers specify constraints (links) between the abstraction and view (the ALV model, Patterson 1991), and the system automatically propagates any changes between them. Similar to Rendezvous' ALV model, the Chiron-1 user interface system has abstract data types (abstractions), dispatchers (links) and views; however, a simpler event-based architecture rather than constraints are used to propagate changes (Taylor, Nies, Bolcer et. al., 1994). While Chiron-1 was not explicitly designed to be a groupware toolkit, a multi-user Tetris game was developed to show the flexibility of its architecture. In Weasel, programmers use a special declarative language called RVL to specify the relations between abstractions and views, how views are customized, and the co-ordination required. As mentioned, GroupKit uses events and environments to co-ordinate views and environments, which probably involves more housekeeping than either ALV or RVL. Again, unlike Rendezvous and Weasel, GroupKit provides other programming abstractions (multicast RPCs) for greater programming flexibility.

Replicated objects. Groupware toolkits based upon object-oriented languages typically promote the idea of object replication as their programming abstraction. For example, ObjectWorld gives application developers a special object called a shareable class. Objects that inherit from this class gain the ability to replicate themselves across application instances, and acquire methods to send and receive messages between replicas. DistView uses a somewhat similar approach to replicate data and views onto that data. Because GroupKit is not object-oriented, it cannot do object replication. We believe replicated objects are a useful programming abstraction and have even built a Smalltalk-based prototype of GroupKit called Gen. However, it is too early to tell if replicated objects should replace the other programming abstractions, or just co-exist with them.

Concurrency control in replicated architectures. Because replicated architectures are distributed systems, consistency maintenance between replicas can be a problem. Some toolkits, such as Share-Kit and earlier versions of GroupKit, have no direct concurrency control while others include some ability to handle it. For example, DistEdit uses atomic broadcasts. ObjectWorld's shareable object have the ability to detect messages that have arrived out of order, and allow programmers to do non-optimistic locking. GroupKit currently supports an optional and rudimentary concurrency control in its environments, and allows developers to add new policies to suit their application. Because we believe that no one concurrency control approach works in groupware (Greenberg and Marwood 1994), environments will be extended to allow programmers to specify the type of concurrency control they want (e.g., optimistic and non-optimistic locking, serialization, etc.).

Centralized architectures. Some toolkits, notably Rendezvous, use a centralized architecture. This greatly simplifies and even eliminates problems that crop up in replicated systems. These include making sure all sites has an up to date copy of the software, and concurrency control. The cost paid for centralization is performance, as the processor becomes the bottleneck when dealing with simultaneous update of many displays. Most other toolkits use some form of replicated architecture, although centralized components may be present (such as GroupKit's Registrar).

Session management. Most toolkits provide only rudimentary and hard-wired session management facilities. Share-Kit, for example, provides only basic connection facilities, although it does allow information about participants and about the session to be transmitted to others upon connection. There have been a few investigations into architectures for flexible session management e.g., Intermezzo (Edwards 1994), but these are not really groupware toolkits. Excepting GroupKit, we are not aware of any other toolkit that lets programmers build both applications and session managers, and that separate the two concepts.

Interface Widgets. While many stand-alone and toolkit-based groupware applications contain ideas that could influence widgets, only a few systems have developed them as reusable multi-user widgets. Smith and Rodden's (1993) "shared interface object layer" SOL is a notable exception (although it is an architectural layer rather than a toolkit). They

considered how shareable versions of single-user widgets such as buttons and text entry fields can be created. Like Shen and Dewan (1992), they concentrated on providing a set of generic access control mechanisms that determine what people could do with these shareable objects. Settable options include who can see it, who can use it, who can move it, and so on. While this kind of capability could (and should) be added to GroupKit, our research interest is towards designing quite different widgets that support particular aspects of group work.

The list above is not complete. For example, toolkit comparisons could have been made contrasting the support provided for: security, privacy and access control; robustness, fault tolerance and performance; capabilities for particular application domains and interface situations; portability and familiarity of the underlying platform and programming language; debugging support; provisions of interface builders, and so on (e.g., see Urnes and Nejabi 1994 for other comparative features).

Virtually all toolkits, including GroupKit, are incapable of handling all the listed features. Most are just prototypes used to explore different ideas, abstractions and architectures. As a result, we do not consider any one toolkit better or worse than another, as it depends upon what the evaluator is looking for. The current set of toolkits are best considered as "breakthrough" systems that will influence the next generation of toolkits.

9. Summary

This paper presented an overview of GroupKit, highlighting the core requirements behind its design and its key features. Its runtime infrastructure automatically manages the creation, interconnection, and communications of the distributed processes that comprise conference sessions. Its programming abstractions allow developers to control the behaviour of distributed processes, to take action on state changes, and to share relevant data. Its set of easy-to-add groupware widgets encapsulate information valuable to conference participants. Its session managers are decoupled from groupware applications and are built by developers to accommodate the group's working style. Examples of both code and GroupKit applications were used to illustrate these features, and to show that the effort required to build groupware using the toolkit is quite reasonable.

While we believe that we are well on the way to understanding what toolkits should provide to groupware programmers, there is still much left to do in GroupKit. In Table 1 and in our original paper on GroupKit (Roseman and Greenberg 1992), we listed a variety of requirements from both the user centered and programmer centered point of view. Some of these remain unimplemented. For example, we still believe that the toolkit should be integrated with other communication channels, such as conventional voice lines (telephones) and digital audio/video (once the format and delivery mechanism becomes standardized and available on all platforms). It should also allow, or at least be capable of invoking, a system for sharing conventional single-user software. The toolkit should present all its capabilities, such as its widgets, its data objects and its graphical objects, as fully group-aware structures; the programmer should not have to waste time co-ercing single user structures into multi-user ones. The toolkit should allow groupware sessions to go beyond the real-time barrier; we should be able to configure electronic meetings and who should be in them ahead of time, and have the meeting artifacts persist beyond a single session (O'Grady and Greenberg 1994). Aside from our own ideas, there are also the many appealing features of the other toolkits described in Section 8.

Because there is too much to do, we are now concentrating on only three major research directions. First, we are continuing our work on widgets, particularly those that support *workspace awareness* (Gutwin and Greenberg 1995; Gutwin, Stark and Greenberg 1995). We define this as the up-to-the-minute knowledge a person requires about other person's interactions with the shared workspace. We believe workspace awareness is required for an individual to co-ordinate and complete their part of a group task. We have developed a preliminary framework that categorizes awareness in terms of how closely users are working together on a task (same or different) and on the view (strict vs. relaxed WYSIWIS), and we have also invented a few prototype widgets that go beyond those illustrated in Section 5. Second, we are investigating alternate metaphors for session management, and whether they can be built easily within GroupKit (Roseman and Greenberg 1994). Example metaphors other than those already shown include allowing people to call one another and supporting awareness of who is around in the community. Third, we are refining our programming abstractions, particular GroupKit's events and environments. We would like to make them even simpler, and are using them to experiment with several concurrency control mechanisms (Greenberg and Marwood 1994).

Groupware toolkits still have a long way to go to catch up to their single-user counterparts. We look forward to the day when all toolkits, perhaps influenced by GroupKit and others in its genre, will incorporate multi-user features. When that day comes, the artificial distinction between constructing single and collaborative systems will disappear.

Availability.

GroupKit is available via anonymous ftp. The release contains all the software, installation instructions, example conference applications and session managers, manual pages, and tutorial documentation. A World Wide Web page also documents the system and work in progress, and a mailing list links the community at large.

site:	ftp.cpsc.ucalgary.ca
directory:	pub/projects/grouplab/software
http:	http://www.cpsc.ucalgary.ca/projects/grouplab/home.html
mailing list:	groupkit-users@cpsc.ucalgary.ca

Caveat. GroupKit is not a commercial product. Its features are not as complete as we would like them to be; bugs appear on occasion, and it can suffer performance problems in certain environments. Although we do not recommend it as a commercial platform, it is an effective tool for: CSCW researchers who wish to examine and extend the ideas presented in this paper; researchers and developers who are prototyping and testing groupware systems, components, and widgets; educators who want to give learners hands-on experience with groupware and its development; and for communities of curious users who wish to experiment with groupware applications.

Acknowledgements.

This research is (gratefully) supported in part by the National Engineering and Research Council of Canada, and by Intel Corporation. Many students at the University of Calgary have contributed in one way or another to GroupKit: Carl Gutwin, Shannon Jaeger, Earle Lowe, David Marwood, Ted O'Grady and Salaam Yitbarek. We also thank users from other sites who have given us feedback and sent us example applications.

References

Baecker, R., Nastos, D., Posner, I. and Mawby, K. (1993) "The User-Centered Iterative Design of Collaborative Writing Software." in *Proceedings of ACM InterCHI'93 Conference on Human Factors in Computing Systems*, April 24-29, Amsterdam, The Netherlands, p399-405, ACM Press.

Beaudouin-Lafon, M. and Karsenty, A. (1992) "Transparency and Awareness in a Real-Time Groupware System." in *Proceedings of UIST'92 Symposium on User Interface Software and Technology*, November 15-18, Monterey, California, p71-80, ACM Press.

Bonfiglio, A., Malatesta, G. and Tisato, F. (1989) "Conference Toolkit: A Framework for Real-Time Conferencing." in *Proceedings of the EC-CSCW '89 First European Conference on Computer Supported Cooperative Work*, September 13-15, Gatwick, London, UK, p303-316.

Cockburn, A. and Greenberg, S. (1995a) "The Design and Evolution of TurboTurtle, a Collaborative Microworld for Exploring Newtonian Physics." Research Report 95/551/03, Department of Computer Science, University of Calgary, Alberta, Canada.

Cockburn, A. and Greenberg, S. (1995b) "TurboTurtle, a Collaborative Microworld for Exploring Newtonian Physics." In *Proceedings of the ACM Conference on Computer Supported Collaborative Learning*, November, in press.

Cockburn, A. and Greenberg, S. (1993) "Making Contact: Getting the Group Communicating with Groupware." in *Proceedings of the ACM COOCS'93 Conference on Organizational Computing Systems,* November 1-4, Milpitas, California, ACM Press.

Dewan, P. (1991) "Flexible User Interface Coupling in Collaborative Systems." in *Proceedings of the ACM CHI'91 Conference on Human Factors in Computing Systems,* April 28-May2, New Orleans, Louisiana, p41-48, ACM Press.

Dourish, P., and Bellotti, V. (1992) "Awareness and Coordination in Shared Workspaces" In *Proceedings of the ACM CSCW Conference on Computer-Supported Cooperative Work*, p107-114, Toronto, Ontario, ACM Press.

Edwards, W.K. (1994) “Session Management for Collaborative Applications. in Conference on Computer Supported Cooperative Work.” in *Proceedings of the ACM CSCW’94 Conference on Computer Supported Cooperative Work*, October 22-26, Chapel Hill, North Carolina, p323-330, ACM Press.

Graham, T.C.N. and Urnes, T. (1992) “Relational Views as a Model for Automatic Distributed Implementation of Multi-User Applications.” in *Proceedings of the ACM CSCW’92 Conference on Computer Supported Cooperative Work,* October 31-November 4, Toronto, Canada, October. ACM Press.

Greenberg, S. (1991) “Personalizable Groupware: Accommodating Individual Roles and Group Differences.” in *Proceedings of the ECSCW’91 European Conference on Computer Supported Cooperative Work,* Amsterdam, The Netherlands, p17-32, Kluwar Academic Press.

Greenberg, S. (1990) “Sharing Views and Interactions with Single-User Applications.” in *Proceedings of the COIS‘90 Conference on Office Information Systems,* April 25-27, Cambridge, Massachusetts, p227-237, ACM Press.

Greenberg, S. and Bohnet, R. (1991) “GroupSketch: A Multi-User Sketchpad for Geographically-Distributed Small Groups.” in *Proceedings of Graphics Interface*, June 5-7, Calgary, Alberta, Morgan-Kaufman Press.

Greenberg, S. and Marwood, D. (1994) “Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface.” in *Proceedings of the ACM CSCW’94 Conference on Computer Supported Cooperative Work*. October 22-26, Chapel Hill, North Carolina, p207-217, ACM Press.

Greenberg, S. and Roseman, M. (1994) “GroupKit.” in *ACM SIGGRAPH Video Review*, Issue 108, ACM Press. Videotape.

Greenberg, S., Roseman, M., Webster, D. and Bohnet, R. (1992) “Human and Technical Factors of Distributed Group Drawing Tools.” *Interacting with Computers*, **4**(3), p364-392.

Gutwin, C. and Greenberg, S. (1995) “Support for Group Awareness in Real-Time Desktop Conferencing.” in *Proceedings of the 2nd New Zealand Computer Science Research Student’s Conference*, April 18-21, University of Waikato, Hamilton, New Zealand.

Gutwin, C., Stark, G. and Greenberg, S. (1995) “Support for Workspace Awareness in Educational Groupware.” in *Proceedings of the ACM Conference on Computer Supported Collaborative Learning, November, in press.*.

Hayne, S., Pendergast, M. and Greenberg, S. (1994) “Implementing Gesturing with Cursors in Group Support Systems.” *Journal of Management Information Systems,* **10**(3), p43-61.

Hill, R.D., Brinck, T., Rohall, S.L., Patterson, J.F. and Wilner, W. (1994) “The Rendezvous Architecture and Language for Constructing Multi-User Applications.” *ACM Transactions on Computer-Human Interaction,* **1**(2), p81-125, June.

Ishii, H. (1990) “TeamWorkStation: Towards a Seamless Shared Workspace.” in *Proceedings of ACM CSCW'90 Conference on Computer-Supported Cooperative Work,* October 7-10, Los Angeles, California, p13-26, ACM Press.

Ishii, H. and Kobayashi, M. (1992) "ClearBoard: A Seamless Medium for Shared Drawing and Conversation with Eye Contact." In CHI '92: Human Factors in Computing Systems, pp. 525-532, Monterey, California, ACM/SIGCHI.

Jahn, P. (1995) “Getting Started with Share-Kit.” Tutorial manual distributed with Share-Kit 2.0.0. Communications and Operating Systems Research Group, Department of Computer Science, Technische Universität, Berlin, Germany. Available via anonymous ftp from ftp.inf.fu-berlin.de in the directory /pub/misc/share-kit. .

Kaplan, S.M., Tolone, W.J., Bogia, D.P. and Bignoli, C. (1992) “Flexible, Active Support for Collaborative Work with Conversation Builder.” *Proceedings of the ACM CSCW’92 Conference on Computer Supported Cooperative Work,* October 31-November 4, Toronto, Canada, p378-385, ACM Press.

Knister, M.J. and Prakash, A. (1990) “DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors.” in *Proceedings of ACM CSCW'90 Conference on Computer-Supported Cooperative Work,* October 7-10, Los Angeles, California, p343-355, ACM Press.

Krasner, G.E. and Pope, S.T. (1988) “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.” *Journal of Object Oriented Programming* **1**(3), August/September.

Linton, M.A., Vlissides, J.M. and Calder, P.R. (1989) “Composing User Interfaces with InterViews.” *IEEE Computer*, **22**(2).

Malone, T.W., Lai, K.Y. and Fry, C. (1992) "Experiments with Oval: A Radically Tailorable Tool for Cooperative Work." *Proceedings of the ACM CSCW'92 Conference on Computer Supported Cooperative Work,* October 31-November 4, Toronto, Canada, p289-297, ACM Press.

O'Grady, T. and Greenberg, S. (1994) "A Groupware Environment for Complete Meetings." in *Companion Proceedings of the ACM SIGCHI'94 Conference on Human Factors in Computing Systems.* Boston, Massachusetts, p307-308, ACM Press.

Ousterhout, J., *Tcl and the Tk Toolkit*. 1994, Addison-Wesley.

Patterson, J.F. (1991) "Comparing the Programming Demands of Single-User and Multi-User Applications." in *Proceedings of the UIST'92 Symposium on User Interface Software and Technology*, November 11-13, Hilton Head, South Carolina, p87-94, ACM Press.

Prakash, A. and Shim, H.S. (1994) "DistView: Support for Building Efficient Collaborative Applications using Replicated Objects." in *Proceedings of the ACM CSCW'94 Conference on Computer-Supported Cooperative Work*, October 22-26, Chapel Hill, North Carolina, p153-164, ACM Press.

Roseman, M. (1995) "When is an Object Not an Object?" in *Proceedings of the Usenix Tcl/Tk Workshop, Toronto, July 6-8.*

Roseman, M. and Greenberg, S. (1994) "Registration for Real Time Groupware." Research Report 94/533/02, Department of Computer Science, University of Calgary, Alberta, Canada.

Roseman, M. and S. Greenberg. (1993) "Building Flexible Groupware Through Open Protocols." in *Proceedings of the ACM COOCS'93 Conference on Organizational Computing Systems,* November 1-4, Milpitas, California, p279-288, ACM Press.

Roseman, M. and Greenberg, S. (1992) "GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications." in *Proceedings of the ACM CSCW'92 Conference on Computer Supported Cooperative Work,* October 31-November 4, Toronto, Canada, p43-50, ACM Press.

Shen, H. and Dewan, P. (1992) "Access Control for Collaborative Environments." in *Proceedings of the ACM CSCW'92 Conference on Computer Supported Cooperative Work,* October 31-November 4, Toronto, Canada, p51-58, ACM Press.

Shepherd, A., Mayer, N. and Kuchinsky, A (1990) "Strudel - An Extensible Electronic Conversation Toolkit." in *Proceedings of ACM CSCW'90 Conference on Computer-Supported Cooperative Work,* October 7-10, Los Angeles, California, p93-104, ACM Press.

Smith, G. and Rodden T. (1993) "Using an Access Model to Configure Multi-User Interfaces." in *Proceedings of the ACM COOCS '93 Conference on Organizational Computing System.* November 1-4, Milpitas, California, p289-298, ACM Press.

Smith, B.C., Rowe, L.A. and Yen, S.C. "Tcl Distributed Programming." in *Proceedings of the Tcl/Tk Workshop*, Berkeley, CA.

Stefik, M., Bobrow, D.G., Foster, G., Lanning, S. and Tatar, D. (1987) "WYSIWIS Revised: Early Experiences with Multi-User Interfaces." *ACM Transactions on Office Information Systems* **5**(2), p147-167.

Tang, J.C. (1991) "Findings from Observational Studies of Collaborative Work." International Journal of Man Machine Studies, **34**(2): p143-160.

Tang, J.C. and Minneman, S.L. (1990) "VideoDraw: A Video Interface for Collaborative Drawing." in *Proceedings of the ACM SIGCHI'90 Conference on Human Factors in Computing Systems,* April 1-5, Seattle, Washington, p313-320, ACM Press.

Taylor, R.N., Nies, K.A., Bolcer, G.A., MacFarlane, C.A., Johnson, G.F. and Anderson, K.M. (1994) "Supporting Separations of Concerns and Concurrency in the Chiron-1 User Interface System." UCI Technical Report 94-12, Department of Computer Science, University of California, Irvine, California, March.

Tou, I., Berson, S., Estrin, G., Eterovic, Y. and Wu, E. (1994) "Prototyping Synchronous Group Applications." *IEEE Computer* **27**(5), p48-56, May.

Trevor, J., Rodden, T. and Mariani, J. (1994) "The Use of Adaptors to Support Cooperative Sharing." in *Proceedings of the ACM CSCW'94 Conference on Computer Supported Cooperative Work*, Oct 22-26, Chapel Hill, North Carolina, p219-230, ACM Press.

Urnes, T. and Nejabi, R. (1994) "Tools for Implementing Groupware: A Survey and Evaluation." Technical report CS-94-03, Department of Computer Science, York University, Toronto, Canada.