

# GroupKit User's Manual

*GroupKit version 3.0 — January 1995*

Mark Roseman  
Saul Greenberg  
Shannon Jaeger

Department of Computer Science  
University of Calgary  
Calgary, Alberta  
Canada T2N 1N4  
Phone: +1 403 220-6087  
Email: [groupkit-bugs@cpsc.ucalgary.ca](mailto:groupkit-bugs@cpsc.ucalgary.ca)

*Draft: January 19, 1995*

# Table of Contents

<b>Introduction</b>	<b>5</b>
About this Manual	5
Previous Knowledge	5
For Further Information	5
<b>GroupKit Overview</b>	<b>7</b>
Run-Time Architecture	7
Registrar	7
Registrar Client	8
Conference Applications	8
Running GroupKit applications	8
Starting a registrar	8
Starting a registrar client	8
Starting the Brainstorm Conference Application	9
Joining an existing	10
Exiting a conference	11
Setting up your .groupkitrc	11
<b>Hello World</b>	<b>12</b>
<b>Sending Messages</b>	<b>14</b>
Message Example	14
Types of Remote Procedure Calls	14
<b>Event</b>	<b>16</b>
Working with events	16
Information about an event	16
Triggers	16
Generating events	17
Ordering of Triggers	17
Events in GroupKit	17
New user events	17
User leaving events	18
Update entrant events	18
Simple Event	18

<b>Environments</b>	<b>20</b>
What are Environments?	20
Using Environments	20
Creating an Environment	20
Adding Information to an Environment	20
Getting Information from an Environment	21
Deleting Information from an Environment	21
Dealing with Hierarchical Information	21
Other Operations	22
Summary of Operations	22
Environments used in GroupKit	22
The “userprefs” environment	22
The “users” environment	23
Notification of Environment Changes	24
Events Generated	24
Example	24
Sharing of Environments	25
Simple Environment Example	25
<b>Class Builder</b>	<b>28</b>
Overview	28
Terminology	28
Class Data Structure	28
Widget Data Structure	29
Creating A Widget Class	30
classname	30
classname_Config	30
classname_ConstructWidget	31
classname_CreateClassRec	31
classname_DestroyWidget	32
classname_InitWidgetRec	32
classname_Methods	32
classname_SetWidgetBindings	32
Using A New Widget Class	32
Known Problems	33
Examples	33
Widget Class fancyLabel	33
Using the FancyLabel Widget Class	35
Creating Widget Class frameLabel	36
Using the FrameLabel Widget Class	37
<b>GroupKit Widgets</b>	<b>39</b>
GroupKit Menu Bar	39

Participant Window	40
Attribute Window	41
About Box	41
Help	42
Topic Window	43
Multi-User Scrollbar	44
Telepointers	45
Gestalt View	47
Functionality	48
Viewport	49
<b>Conference Application</b>	<b>51</b>
Example: Brainstorming	51
Initializing the Conference Application	51
Building the Menus	51
Building the Main Interface	52
Broadcasting Changes	53
Updating New Entrants	54
Odds and Ends	54
Example: Drawing Program	55
<b>Registration</b>	<b>59</b>
Overview of the Registration Architecture	59
The Role of the Registrar	59
The Role of the Registrar Client	59
Implementation Concepts	60
Sending Messages to the Registrar	61
Registration Environments	63
The “confs” environment	63
The “spawned” environment	63
Registration Events	64
Changes to Conferences	64
Changes to Users	65
Events about Conference	67
Supplied Event Handlers	67
Summary of Environments, Events and Handlers	69
Miscellaneous Commands	70
A User Interface to Create Conference	70
Creating Conference	70
Keeping Track of Joined Conference	71
Creating an Originator ID	71
Putting it All Together	71

Initializing the Registrar Client	71
Specifying Registration Policy	71
Building the Interface	73

<b>Index</b>	<b>79</b>
--------------	-----------

# Introduction

GroupKit is a toolkit for developing real-time groupware applications. Based on Berkeley's freely available Tcl/Tk language, it provides the basis for building groupware applications. Tcl is an interpreted shell-type programming language, and Tk is an interface toolkit for the X11 window system. GroupKit also relies on Tcl-DP, a Tcl front-end to standard Unix sockets, for its communication needs. GroupKit programs, therefore, run on Unix environment.

---

## About this Manual

This tutorial is an introduction to developing groupware applications using GroupKit. Having completed this tutorial, the reader should be able to run and develop groupware applications using many of the important concepts and components provided by the toolkit.

---

## Previous Knowledge

This manual assumes a familiarity with Tcl/Tk, as well as some knowledge about groupware systems and their features. Knowledge of the Tcl-DP extension is not required unless you are dipping into GroupKit's source code. The manual also assumes that Tcl, Tk, TCL-DP, and GroupKit have been installed on your Unix system. If these systems have not been installed, see the README file in the GroupKit distribution.

---

## For Further Information

The README file in the GroupKit distribution provides details for installing GroupKit on your Unix system. You should read this if you are installing and/or maintaining GroupKit. Also see the README's in the `reg_clients` directory and the `classBuilder` directory for more information on different registration techniques and constructing widgets respectively. Also see the man pages that have been included with the distribution. They contain specific information for the various aspects of GroupKit.

GroupKit also has a home on the world wide web,

**<http://www.epsc.ualgary.ca/projects/grouplab/projects/groupkit>**

Here you will find information about GroupKit, lots of sample applications, and links to related material.

*GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications*, (Proceedings of the 1992 ACM Conference on Computer Supported Cooperative Work—CSCW'92, Toronto, Ontario) describes an earlier version of GroupKit designed using C++ and InterViews, a C++ interface toolkit. It illustrates some of the important concepts in GroupKit.

*Building Flexible Groupware Through Open Protocols*, (Proceedings of the 1993 ACM Conference on Organizational Computing Systems—COOCS '93), describes the *open protocols* method of groupware design upon which GroupKit relies.

*Tcl/Tk as a Basis for Groupware*, (Proceedings of the 1993 Tcl Workshop, Berkeley, CA), explains the benefits of Tcl/Tk for groupware development.

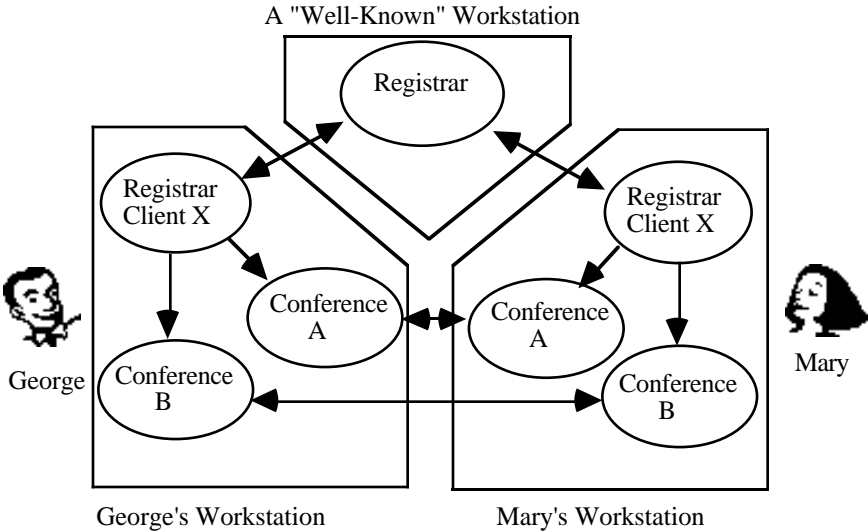
*An Introduction to Tcl/Tk*, (published by Addison-Wesley), by John K. Ousterhout is a learning guide to Tcl/Tk.

Documentation on the Tcl-DP extension can be found in its on-line manual pages which should have been installed with Tcl-DP.

# GroupKit Overview

## Run-Time Architecture

GroupKit applications consist of a number of mostly replicated processes arranged across a number of machines. The following diagram illustrates an example of the processes running when two people are communicating to each other through two conferences 'A' and 'B'. Each person would see windows representing each conference on their screen, as well as a window representing their Registrar Clients (which may be different!). Processes are represented by ovals, and the lines joining them indicate communication paths.



### Registrar

The central Registrar runs on a “well-known” workstation, and is the contact point that all processes use to find out what conferences are around. It is started by executing *registrar*. The Registrar maintains a list of all the conferences active on the system and a list of all the conference users. The Registrar itself does not have a policy on how conferences are created or deleted nor on how users join or leave conferences. There is usually one Registrar for your entire community of conference users. The diagram above illustrates this.

## Registrar Client

The Registrar Client (one per user) allows users to create, delete, join or leave conferences. It interacts with other Registrar Clients through the central list provided by the Registrar. The Registrar Client provides both a user interface as well as a policy dictating how conferences are created or deleted and how users are to enter and leave conferences. Different Registrar Clients can be created to suit different registration needs. Several different Registrar Clients are provided in the GroupKit package. Developers can create new registrar clients if desired, although this is not covered by this tutorial. The diagram above shows how each person has a registrar client on their workstation.

## Conference Applications

The conference application is the actual shared application built using GroupKit, and is separate from the registration system. A conference application is typically a groupware tool like a shared editor, whiteboard, and so on. The GroupKit application developer creates these conference applications.

The diagram above illustrates these concepts. Conference A (perhaps a shared whiteboard) is made up of two replicated processes, one belonging to Registrar Client X running on George's machine and seen by George, and the other to Registrar Client Y that runs on Mary's machine and seen by her. These conference processes maintain communication facilities necessary for exchanging messages with each other. The user interface portions of groupware applications use these facilities extensively.

---

## Running GroupKit applications

There are several steps in running a GroupKit application. You must first configure your system via the *.groupkitrc* file. You must then start a registrar (if one is not already running at your site), and then a Registrar Client. Using the Registrar Client, you can create and join any number of GroupKit conferences.

### Starting a registrar

The first step is to create a registrar process. This is done only once on the designated machine. The machine name should be the same one set in the *.groupkitrc* in the variable "registrar host". You normally have only one registrar process that is used by all your GroupKit users. Do this by logging on to that machine and entering

```
registrar &
```

If you wish to override the port value by a command-line options as follows:

```
registrar -p9999 &
```

### Starting a registrar client

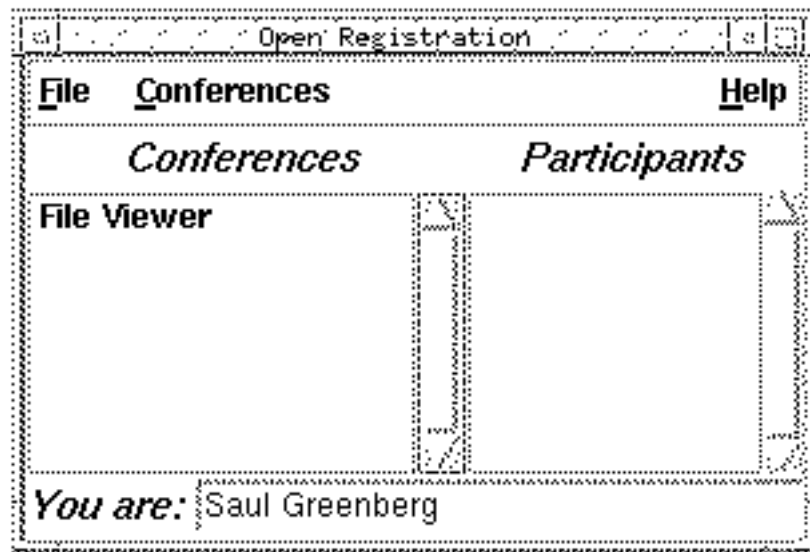
Next, open a registrar client. We shall use *open.reg*, the basic registrar client supplied in GroupKit. Start *open.reg* by entering

```
open.reg &
```

The host and port values for the registrar to connect to can be specified with command-line options such as:

```
open.reg -p9999 -hmanaslu &
```

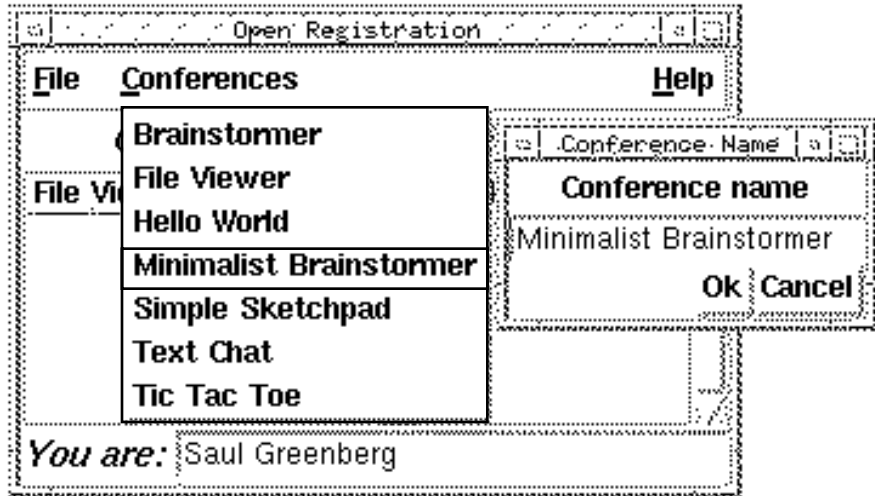
Either of these brings up window below. For illustration, we show it with one conference application called “File Viewer” running; your own conferences window will probably be empty for now. Your name is shown on the bottom, and you can change this if you wish. The name is the same one set in the `.groupkitrc` file.



There are other registration application included in GroupKit. These include a "rooms-based" system and the other system is designed for use in a formal facilitated meeting. To find out how to use these see the README file in the `reg_clients` directory.

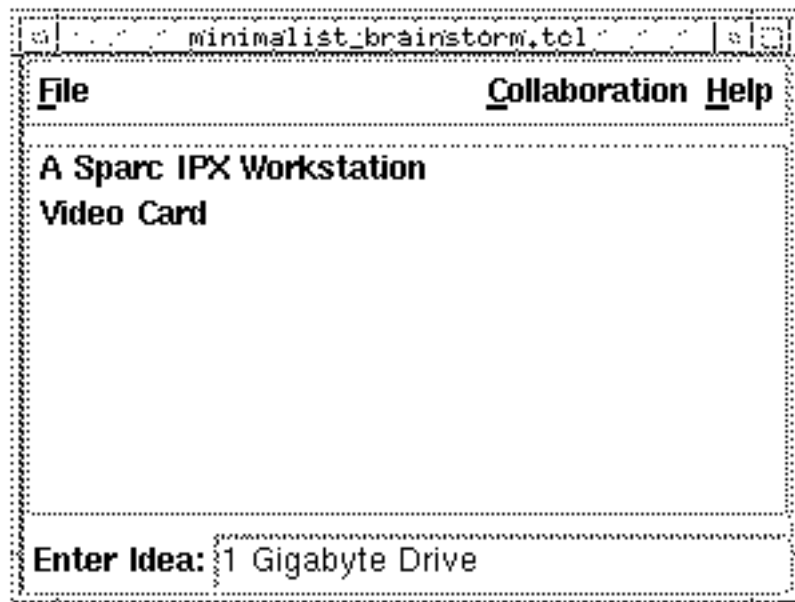
## Starting the Brainstorm Conference Application

To start a new conference, pull down the “Conferences” menu and choose one of the listed conference applications (the list of applications are specified in the `.groupkitrc` file). When you select one, e.g., “Minimalist Brainstormer”, a dialog box will ask you to confirm; it will also let you optionally change the name of the conference from the application name to something that is more meaningful to the targeted group e.g., “Equipment Purchase Ideas”. Here is what the “Conferences” pull-down menu and the dialog box look like:



This starts up *Minimalist Brainstormer* conference, whose code was supplied with GroupKit in *confs/minimalist\_brainstorm.tcl*, that will run in a new window, similar to the one below (although this one shows it after a few "ideas" have been entered). To enter ideas, just type into the Enter Idea field. When you hit the RETURN key, the idea will be added to the shared list shown in the middle of the window.

The name of the conference is now displayed in the "Conferences" listbox in the open.reg window.



## Joining an existing conference

At this point, you are just running the Minimalist Brainstormer as a single-user application. Someone else can create an open.reg registrar client on their

machine, or you can pretend that there are two users in the conference by creating another open.reg on your own machine.

Click once on Minimalist Brainstormer in the Conferences list (not the menu!) to display the list of users of that conference in the “Users” listbox. Double-clicking on the item will automatically join that conference.

There are now two replicated processes running as a “Minimalist Brainstormer” conference. Ideas may be entered from either process and they will appear in both shared lists.

## Exiting a conference

To exit from a conference, choose “Quit” under the “File” menu. This eliminates the local conference process. To see who is in the conference select “Show Participants” in the “Collaboration” pull-down menu. Finally you can get information about GroupKit and help on this application by selecting items from the “Help” pull-down menu.

## Setting up your .groupkitrc

You should have a file called *.groupkitrc* in your home directory, If you do not already then one will be created automatically for you. A sample file is displayed below. You may have to change some of this information; use the comments as a guide. Note that “\$gk\_library” is a variable whose value is based on the value of the GK\_LIBRARY environment variable.

```
# Set this to your name
userprefs name "Mark Roseman"

# Set this to your internet domain
userprefs internetdomain ".edm.isac.ca"

# Set this to the computer that will run the registrar
registrar host isasun-1

# Set this to your favorite color---different users can
# have different colors!
userprefs color red

# we want debugging output on all events
userprefs debugevents 1

# the lines below specify which conferences we know about
userprefs "prog.Brainstorming Tool.cmd" \
    "exec gkwish -f
    $gk_library/confs/brainstorm.tcl"

userprefs "prog.Simple Sketchpad.cmd" \
    "exec gkwish -f
    $gk_library/confs/simple_sketchpad.tcl"

userprefs "prog.Text Chat.cmd" \
    "exec gkwish -f $gk_library/confs/text_chat.tcl"
```

# Hello World

Now that you know something about conferences show you how to build some of your own. The Hello World conference application is the only hello world application that really allows you to say “Hi” to the entire world. It is a good program for seeing some really basic groupware code, but it is not powerful enough to be really useful. Later examples, in the Conference Applications chapter, will formally introduce terminology and show more complex applications.

Before going on, try running the *Hello World* conference application with several participants (see previous chapter for an explanation on how to run a conference). As you will see, *Hello World* includes the GroupKit default menu and associates some help with the *Help* pulldown menu, and displays a button labeled “Hello World” (the first screen below). When you press the button, the button label changes on all participants displays (including yours) to “Saul Greenberg says hello!” (or whatever your name is) (the second screen below). This is truly a hello world program!



*Hello world*, like all conference applications, begins with the procedure call `gk_initConf $argv`. This call establishes the run-time infrastructure around the application. We also include a standard Tk instruction to the window manager that makes the window resizable. We then add in GroupKit’s default menubar which can be obtained by way of a call to `gk_defaultMenu`, with the parameter `.menubar` which will become the menu bar’s widget name.

```
wm minsize . 250 50
gk_initConf $argv
gk_defaultMenu .menubar
pack .menubar -side top -fill x
```

The next step is to add help to the Help pulldown menu. First we define a help title and some formatted help text (see the widget section for an explanation of how help formatting is defined and how to add to the menubar).

```

set help_title "About Hello World"
set help_text {
    {normal} {Press the button to say hello to all
conference participants.}
}

.menubar itemcommand 2 add command \
    -label "$help_title" \
    -command {gk_topicWindow .helpWindow \
        -height 8 \
        -width 20 \
        -title "$help_title" \
        -text "$help_text"}

```

Two messages are then defined. The standard message is just “Hello World”. The second greetings message gets the name of the local user by the call **gk\_getLocalAttrib** with the argument **username**, and concatenates it with the string “says hello”.

```

set standard_message "Hello World"
set greetings_message [concat [gk_getLocalAttrib
username] "says hello!"]

```

We then define a callback, called `say_hi`, that will briefly change the label in the button from “Hello World” to the one defined in the greeting message.

```

proc say_hi {new_message} {global standard_message
    .hello configure -text $new_message
    after 1500 {.hello configure -text $standard_message}
}

```

The button is then created. The novel thing about it is its command: **gk\_toAll**, which tells all participant to execute the command the procedure `say_hi`, whose argument is the greeting message containing the local user’s name. This means that when one person (say George) presses the button, all instances of the application will execute the callback `say_hi` “George says hello”, and the button is changed throughout.

```

button .hello \
    -text $standard_message \
    -command "gk_toAll say_hi [list $greetings_message]"
pack .hello -side top

```

# Sending Messages

GroupKit relies heavily on a form of remote procedure call to communicate and share information between different conference processes. GroupKit's messages work similar to the "send" command in Tk — but rather than sending a command only to a program running on the same workstation display, a command is sent to a user running a conference process on another workstation. Messages are executed on the remote workstation as if they were initiated on that machine.

---

## Message Example

The Brainstorming conference presented later gives a good picture of how sending messages can easily turn a single user application into a multi-user application. Imagine the following procedure is called everytime we want to insert a new idea into our list (this is the single user case):

```
proc insertIdea idea {
    .idealist insert end $idea
}
```

This would insert the idea into the list called `.idealist`. Now lets see the multi-user version of this:

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

The difference here is that not only do we insert the idea into our own list, but we send a message to every other conference process to insert the idea into their own lists. A simple mechanism, but it provides a very straightforward and powerful distributed programming method.

---

## Types of Remote Procedure Calls

GroupKit provides three different remote procedure call facilities, all which send commands to be executed on workstations in the conference. They differ in who the messages are sent to.

The *gk\_toAll* command sends a command to all users in the conference, including the local user. As you saw in the above example, this will result in the same command being executed everywhere.

The *gk\_toOthers* command sends a command to all the remote users in the conference, but does not execute the command on the local workstation. This is useful when you want to send a message to every other user but not yourself. Its syntax is similar to *gk\_toAll*:

```
gk_toOthers "puts hello"
```

Finally, *gk\_toUserNum* sends a command to a single user in the conference, identified by their unique user number (see the section on Environments to find out how to determine a user's unique user number). Again the syntax is similar:

```
gk_toUserNum1 $usernum "puts hello"
```

---

<sup>1</sup> There is one additional command, *gk\_RPCtoUserNum*, which works like *gk\_toUserNum* except that it waits for the command to finish executing on the remote machine and returns the result to the local machine. However, in general it is not a good idea to use this, because slow networks can result in long delays and a sluggish application. It is almost always better to specify a "request/response" pair of procedures which lets the caller carry on after a request, and later receive the result in a separate response.

# Events

Another abstraction used in GroupKit is the *event*, which provides a notification mechanism, such as for a new user joining the conference. This section explains how to use events and describes the events that are generated within GroupKit conferences.

---

## Working with events

Events are used for notification when things happen, and provide a mechanism, similar to callbacks used by interface widgets, to allow action to be taken on the event. Dealing with events requires working with two pieces — the event itself, and a trigger.

### Information about an event

An event consists of a set of attribute/value pairs which provide information about the event. All events include an attribute called “type” which identifies the kind of event. The other attributes depend on the particular event. For example, an event of type “updateEntrant” (which specifies that the local user should provide all the existing conference state information to a new user) contains an attribute of type “usernum” specifying the unique user number of the new user.

The attributes of an event — during the time the event is active — can be examined using the *gk\_event* command. For example, to determine the type of event, you could use the following:

```
gk_event type
```

### Triggers

In order to receive events, you need to specify an event handler or *trigger*. A trigger specifies what events you are interested in, and what action to take if the event is generated. Triggers are specified using the *gk\_on* command, and take the form “gk\_on pattern action”. You can think of a trigger as an “if” statement that isn’t actually executed until an event is generated. For example, the following trigger might be used to respond to the updateEntrant event described earlier:

```
gk_on {[gk_event type]==“updateEntrant”} {  
    helpTheNewGuy [gk_event usernum]  
}  
  
proc helpTheNewGuy usernum { ... }
```

GroupKit maintains a list of all the triggers described by `gk_on`. When an event is generated, all triggers are examined, and those whose pattern match are executed.

## Generating events

GroupKit will generate a number of events for you (described below), but you may also choose to generate other events particular to your application and write your own triggers for those events. Events are a handy way to communicate between different parts of the same program, such as an underlying data structure and a widget providing a view of that data structure.

To generate an event, use the `gk_postEvent` command. The command takes as its argument a keyed list specifying the event's attributes and their values. All events must have an attribute called "type." For example:

```
keylset event type playGame which SimCity
gk_postEvent $event
```

## Ordering of Triggers

Because the action portions of a trigger usually has side effects, specify triggers carefully. When generating an event, GroupKit examines triggers in the same order that they were declared using the "gk\_on" command — triggers specified earlier will be fired earlier. Be aware of potential conflicts when specifying triggers.

For example, you may set up two triggers to fire on an event signaling a user leaving a conference. If the first trigger deletes all information about that user, and the second trigger displays a message saying that user is leaving, the second trigger had better not need any of the information deleted by the first trigger. If it does, the second trigger should be specified first, so that the information will not yet have been deleted when it fires.

---

# Events in GroupKit

GroupKit conferences generate a number of events to notify changes in a conference. There are events for new users joining, users leaving, and updating newcomers.

## New user events

Whenever a new user arrives in the conference, GroupKit generates a *newUserArrived* event. By setting a trigger on this event, special actions can be taken whenever anyone new joins. The event has only one additional attribute:

---

type	newUserArrived
usernum	user number of the newly arrived user

---

At the point the event is generated, the information on the new user is already present in the users environment (see the following chapter on Environments).

## User leaving events

Whenever a user leaves the conference, GroupKit generates a *userDeleted* event. By setting a trigger on this event, you can take special action whenever anyone leaves. Again, the event has only one additional attribute:

---

type	userDeleted
usernum	user number of the deleted user

---

At the point the event is generated, the information on the user has not yet been deleted from the users environment. It will be deleted immediately after all triggers are given a chance to respond to the event (see the following chapter on Environments).

## Update entrant events

When users first enter a conference, they do not have the information generated by the other participants (for example, the list of ideas in a brainstorming conference or the drawing in a sketchpad conference). GroupKit provides a mechanism to bring new users up to date with what has previously happened in the conference, if required by your application.

GroupKit chooses one of the existing users to update the new user. That user's conference process receives an *updateEntrant* event, specifying the user to be updated:

---

type	updateEntrant
usernum	user number of the user who needs updating

---

---

## Simple Event Example

This example demonstrates how standard events can be traced and how to create new ones. It creates a visible "stack" of event messages, actually just a listbox with messages inserted at the top. A button lets you post an application-specific event.

*Event Example*, like all conference applications, begins with the procedure call **gk\_initConf \$argv**. This call establishes the run-time infrastructure around the application. Also a standard Tk instruction to the window manager that makes the window resizable is included. Then add in GroupKit's default menu bar which can be obtained by way of a call to **gk\_defaultMenu**, with the parameter **.menubar** which will become the menu bar's widget name. The application window, a "Message Stack" that displays events, and a button for incrementing a counter are both constructed. Notice that the command for the button uses the *gk\_toAll* and *gk\_postEvent*. This causes the event to be posted to all the user's in the conference.

```
gk_initConf $argv
wm minsize . 60 10
gk_defaultMenu .menubar

listbox .message_stack
```

```

.message_stack insert 0 "Bottom of Message Stack"

button .postbutton \
    -text "Post a Hello event" \
    -command {gk_toAll gk_postEvent $my_event}
pack .menubar .message_stack .postbutton -side top \
    -fill x

```

The little procedure below prepends the sender's name to the message. The number of the user who caused the event to occur is used to determine the user's name. If the local user is sending the message then check what is stored in the local users environment for otherwise check the remote users list. Then concatenate the users name, user's number and the note and display it in the message stack.

```

proc showMessage {number note} {
    if {$number == [users local.usernum]} {
        set name [users local.username]
    } else {
        set name [users remote.$number.username]
    }
    .message_stack insert 0 \
        [concat $name "-" $number "-" $note]
}

```

Now the **Hello** event is defined which is posted whenever the button is pressed. It is a keyed list that is stored in the variable *my\_event*. The keys in the list are type, usernum and message.

```

keylset my_event type hi_there usernum \
    [users local.usernum] message "says Hello"

```

Finally a number of event handlers. All of them use the showMessage procedure to add the current event to the message stack.

```

gk_on {[gk_event type] == "newUserArrived"} {
    showMessage [gk_event usernum] "arrived"
}

gk_on {[gk_event type] == "userDeleted"} {
    showMessage [gk_event usernum] "left"
}

gk_on {[gk_event type] == "updateEntrant"} {
    showMessage [gk_event usernum] "wants updating"
}

gk_on {[gk_event type] == "hi_there"} {
    showMessage [gk_event usernum] [gk_event message]
}

```

# Environments

GroupKit makes heavy use of a data structure called an environment. Environments serve to hold pieces of related information within a single data structure. This section describes how environments work and the environments used by GroupKit. The section also describes how you can easily share environments between different users in a GroupKit conference.

---

## What are Environments?

Environments are *hierarchically structured dictionaries*, which serve as collections of related information. For example, in GroupKit conferences there is an environment holding all the information about users in the conference.

Information in environments is accessed via a *key*, which specifies where within the environment information is stored. Because environments are hierarchical, keys serve to encode this hierarchy. Keys use a period (“.”) as the hierarchy delimiter. For example, the key “local” refers to information at the first level of the hierarchy, whereas the key “local.name” refers to information deeper in the hierarchy.

---

## Using Environments

Environments are accessed using an “object oriented” syntax, similar to that used in Tk widgets. Creating an environment has the side effect of creating a command (with the environment’s name) that is thereafter used to manipulate the environment.

### Creating an Environment

Environments are created with the “gk\_newenv” command, passing the environment name as a parameter. For example:

```
gk_newenv users
```

### Adding Information to an Environment

To add information to an environment, you need to specify a *key* and a *value*. The key specifies where in the environment the information is to be stored, while the value specifies the information to store. As with all operations on environments, adding information uses the name of the environment (“users” in the above example) as the command. For example:

```
users local.name "Mark Roseman"  
users remote.1.name "Saul Greenberg"
```

## Getting Information from an Environment

Extracting information from an environment is similar. We again provide the key specifying where in the environment the information exists. To retrieve the previous pieces of information we would use:

```
users local.name  
users remote.1.name
```

Because environments are hierarchical, it can be important to find out what elements exist in the hierarchy. This is done using the “keys” subcommand on an environment. For example, assuming the above encoding, we may wish to know the user numbers of the remote users (this is the “1” in “remote.1.name”). The following code would provide us with a list of these keys:

```
users keys remote
```

## Deleting Information from an Environment

To delete a particular piece of information, we use the delete subcommand, providing the key specifying what information we want to delete. This will delete the information at the key and also all information further down the hierarchy from the key. So for example:

```
users delete remote.1.name  
users delete remote.2
```

## Dealing with Hierarchical Information

Because information in environments is represented hierarchically, several of the operations work on entire hierarchies. You’ve already seen this with the delete operation. When you retrieve information representing a hierarchy, it is returned as a data structure called a *keyed list*. For example, executing the command:

```
users remote.1
```

may result in something like the following being returned:

```
{name {Saul Greenberg}} {userid saul} {host janu}
```

As a convenience, there is an *import* operation to aid in inserting a keyed list into an environment. Its use is preferred over adding the entire list directly. The import operation takes a key specifying where the list is to be imported and a keyed list specifying the data to import—and the data’s structure. For example:

```
keylset saulinfo name Saul host pumori port 9156  
users import remote.1 saulinfo
```

Note that the “keys” subcommand discussed above provides a useful way to navigate through an environment; by recursive calls at each level, the entire hierarchy can be traversed. More typically you’d use it to retrieve information spread out across the environment, such as the following script which generates a list containing the names of all remote users:

```
set namelist ""
```

```
foreach i [users keys remote] {
    lappend namelist [users remote.$i.name]
}
```

## Other Operations

To destroy an environment, the “destroy” subcommand can be used, for example:

```
users destroy
```

To display the contents of the environment for debugging purposes, the “debug” subcommand can be used:

```
users debug
```

## Summary of Operations

---

<code>gk_newenv env</code>	create an environment
<code>env key val</code>	add information to the environment
<code>env key</code>	extract information from the environment
<code>env keys [key]</code>	find all keys below the given level
<code>env delete key</code>	delete information from the environment
<code>env import key keylist</code>	import a hierarchy of data into the environment
<code>env destroy</code>	destroy the environment
<code>env debug</code>	print the environment on stdout

---

---

## Environments used in GroupKit

Environments are used throughout GroupKit. When building GroupKit conferences, you will quickly encounter two of them — *userprefs* and *users*.

### The “userprefs” environment

In GroupKit, user’s preferences are stored in an environment called *userprefs*. It holds information such as your name, internet domain, the GroupKit conferences that you know about, and so on. Most of this information comes from your preferences file, *.groupkitrc*. Below is a fragment from this file indicating the types of information that are normally stored in this environment.

```
# Set this to your name
userprefs name "Mark Roseman"

# Set this to your internet domain
userprefs internetdomain ".edm.isac.ca"

# Set this to your favorite color
userprefs color red

# GroupKit conferences we have available
```

```

userprefs "prog.Brainstorming Tool.cmd" \
  "exec gkwish -f $gk_library/confs/brainstorm.tcl"

userprefs "prog.Simple Sketchpad.cmd" \
  "exec gkwish -f $gk_library/confs/simple_sketchpad.tcl"

userprefs "prog.Text Chat.cmd" \
  "exec gkwish -f $gk_library/confs/text_chat.tcl"

```

## The “users” environment

GroupKit holds information on all the users in a conference in an environment called *users*. This environment stores the name, host, socket connection and so on for all users. It also provides some information about the conference as a whole.

Users in a conference are referenced by a unique user number, assigned by the registrar. There are two types of users in a conference, *local* and *remote*. The local user is the user running the conference process, while remote users are users at other workstations, running their own conference processes, but connected to the local user’s process.

In the users environment, information on the local user is kept underneath the key “local”, whereas remote users are kept under the key “remote”, and further divided by their user number. So for example, if Mark is the local user, and Saul and Shannon are remote users connected to Mark, the user environment might contain information similar to the following:

```

users local.username is Mark
users remote.1.username is Saul
users remote.2.username is Shannon

```

Generally, the following information is provided about each user in the conference, whether local or remote:

---

<i>prefix</i> .userid	the login userid
<i>prefix</i> .host	the host name running the process
<i>prefix</i> .port	the port number of the user’s Registrar Client
<i>prefix</i> .username	the user’s full name if available

---

In addition, the following information is available about the local user and the conference itself:

---

ocal.usernum	the user number of the local user
ocal.confnum	the unique number of the conference
local.regghost	the Registrar Client of this user
local.regport	the Registrar Client of this user

---

You can add additional information to this environment. For example, if the conference needs to keep track of Saul’s activity level, you could do:

```

users remote.1.activity 10

```

---

# Notification of Environment Changes

You can request that environments generate events when they are changed. This is useful if several parts of a program must react to changes in an environment. A typical use would be a drawing program, where an environment keeps track of the size and position of underlying objects. When the environment changes, another part of your program can be notified so that it will update a canvas window displaying these objects. Such separation of a view of an object from its underlying representation is a common design choice in building groupware systems.

To request that the environment should generate events when it changes, use the “-notify” flag when creating the environment, for example:

```
gk_newenv -notify drawobjs
```

## Events Generated

Three events are generated, when information is added to, changed, or deleted from the environment. All these events contain the same attributes: the environment that is changing, and the key which identifies where in the environment the change is happening. This is summarized below

---

Event Type	addEnvInfo
	changeEnvInfo
	deleteEnvInfo
Other Attributes	env
	key

---

## Example

The following brief example illustrates how to use the events. A more detailed example can be found later in the manual under the “Drawing Program” section.

```
gk_newenv -notify testing

gk_on {[gk_event type]=="addEnvInfo"}&& \
      ([gk_event env]=="testing") {
  puts "new item: [gk_event key] -> [testing $key]"
}

gk_on {[gk_event type]=="changeEnvInfo"}&& \
      ([gk_event env]=="testing") {
  puts "changed item: [gk_event key] -> [testing $key]"
}

gk_on {[gk_event type]=="deleteEnvInfo"}&& \
      ([gk_event env]=="testing") {
  puts "deleted item: [gk_event key]"
}

testing item1 first_value
testing item1 second_value
testing delete item1
```

---

## Sharing of Environments

Environments can also be shared among the users in a conference. In a shared environment, when any user makes a change to the environment, the change is sent to all the other users as well. Combined with the notification of changes discussed above, shared environments can provide a very useful way to implement such groupware abstractions as What You See Is What I See (WYSIWIS) views on a shared data structure.

*Note: Currently shared environments do not implement any concurrency control — changes are multicast to users with no guarantee that changes will arrive in the same order. In the future, shared environments will be expanded to include various types of concurrency control, such as serialization of changes through a central “owner” for the environment, locking, and more sophisticated schemes.*

To request that an environment be shared, specify the “-share” option when creating the environment. This can also be combined with the “-notify” option, for example:

```
gk_newenv -notify -share drawobjs
```

Combining these two options yields a powerful abstraction. The “Drawing Program” example shown later, as well as the Hypertext Browser in the GroupKit distribution, both advance from single-user to multi-user programs merely by adding the “-share” flag to their environments. Of course, both programs were originally structured so as to maintain a view on an underlying data structure, but by adopting this paradigm where appropriate, much of the data sharing work can automatically be done by GroupKit.

*Note: In order to update new entrants to the conference use **auto\_update**. This command will create an event for the receiver. The type of the event is **envReceived** and then **env** key is the name of the environment. This event indicates that they have been shipped over an entirely new set of information.*

---

## Simple Environment Example

This is a rather simple example which demonstrates the usage of environments in GroupKit. In particular how standard environments can be queried, new ones created, and shared. It also creates a display of:

- several existing environment statistics,
- a user created date which is updated every few seconds
- shared environment containing a global counter

*Environment Example* like all conference applications, begins with the procedure call **gk\_initConf \$argv**. This call establishes the run-time infrastructure around the application. A standard Tk instruction to the window manager that makes the window resizable. Then add in GroupKit’s default menubar which can be obtained by way of a call to **gk\_defaultMenu**, with the parameter **.menubar** which will become the menu bar’s widget name. The application window, a “Message Stack” that displays events, and a button for incrementing a counter are then created.

```
gk_initConf $argv
wm minsize . 60 20
```

```

gk_defaultMenu .menubar

listbox .message_stack
button .counter_button -text "Increment Global Counter" \
    -command {shared_env globalcounter \
        [expr 1+[shared_env globalcounter]]}

pack .menubar -side top -fill x
pack .message_stack -side top -fill both -expand t
pack .counter_button -side top

```

The next procedure displays a number of items that are in the environment. It gets the information from the `local_env`, `shared_env`, `registrar`, and `users` environments. It uses the following procedure, `insertInfo`, to actually insert the information into the listbox. This routine is called whenever any of the environment data gets changed.

```

proc displayEnvironment {} {
    .message_stack delete 0 end
    insertInfo .message_stack [local_env date] ""
    insertInfo .message_stack "Global Counter:" \
        [shared_env globalcounter]
    insertInfo .message_stack "Connections" ""
    insertInfo .message_stack "- Internet:" \
        [userprefs internetdomain]
    insertInfo .message_stack "- Registrar:" \
        [registrar host]
    insertInfo .message_stack "Local Participant" ""
    insertInfo .message_stack "- Number:" \
        [users local.usernum]
    insertInfo .message_stack "- Name:" \
        [users local.username]
    foreach number [users keys remote] {
        insertInfo .message_stack "Remote Participant" ""
        insertInfo .message_stack "- Number:" \
            [users remote.$number.usernum]
        insertInfo .message_stack "- Name:" \
            [users remote.$number.username]
    }
}

proc insertInfo {w message content} {
    $w insert end [concat $message $content]
}

```

The next little procedure updates the environment `local_env` every 3 seconds to contain the current date in `date`. The lines below it create the `local_env` environment and then the date updating is started by calling `setDate`. Notice that this environment is not shared so each user can have a different date, depending on what their UNIX `date` command returns.

```

proc setDate {} {
    local_env date [exec date]
    after 3000 setDate
}

gk_newenv -notify local_env
setDate

```

The following lines create a shared environment. This environment will contain the same information for all users. This environment that contains a global

counter that all users can update. Also the information can be propagated as is done in this example.

```
gk_newenv -notify -share shared_env
shared_env globalcounter 0
```

Finally we have a number of event handlers. The first one updates the display when the shared environment, `share_env`, gets changed. The next one updates the display whenever the date, `local_env`, gets changed.

```
gk_on {( [gk_event type] == "changeEnvInfo") && \
      ( [gk_event env] == "shared_env") } {
    displayEnvironment
}

gk_on {( [gk_event type] == "changeEnvInfo") && \
      ( [gk_event env] == "local_env") } {
    displayEnvironment
}
```

# Class Builder

This chapter presents the GroupKit class builder. In particular it describes how it can be used to create new widgets. The widgets created for use by GroupKit applications are discussed in the next chapter.

---

## Overview

The intention of the GroupKit class builder is to make building combination widgets<sup>2</sup> easier and to make them behave like a single widget. Although this application is included in the GroupKit distribution, it is a standalone program requiring Tcl 7.3 and Tk 3.6. In version 3.0 of GroupKit nearly all of the widgets included in the package use the class builder, which are good examples of complicated widgets. Also the *myLabel* widget is used as a running example through the discussion which is followed by two more examples. All of the examples can be found in the class Builder directory.

One thing to keep in mind throughout this discussion is what it means to be a combination widget constructed by this class builder. What really happens is that the class builder created a frame widget. Then all of the widgets specified are constructed inside of this frame. Thus, the frame can be considered the “root widget” of the combination widget. There are mechanisms provided which allow this frame widget to be manipulated.

## Terminology

Throughout this discussion, the term **classname** refers to the name of the new class of widgets that is being created. For example, *classname(inherit)* is one of the items that is part of the data structure. Thus, if the new class being created was called *myWidget*, then *myWidget(inherit)* would be part of its data structure. **Data** is the name of the widgets data structure. Finally, **root widget** refers to the frame widget that the combination widget is built inside of.

## Class Data Structure

This is one of the two basic data structures used by the class builder, it defines attributes for the class of widgets. This array is only created once and the values are stored in the global array **classname**.

---

<sup>2</sup> Combination widgets are composed of two or more standard Tk widgets.

### **classname(inherit)**

Denotes which class(es) a particular widget class inherits its options and commands from. What is inherited is based on the order the classes appear in the list; the first one in the list is given highest priority. Any widget class defined by the class builder or any standard Tcl/Tk classes are valid entries. *Note that how the commands and options are used is left up to the designer of the new class, a future release may improve on this.*

### **classname(rootOptions)**

The options that are not specific to the class itself. Typically, contains the inherited options. There usually is no need to add to the list since by specifying the inheritance this will be done automatically.

### **classname(options)**

The options that are unique to the particular class of widgets; these are not inherited. Note that every option in this list must be defined, see below for more information.

### **classname(methods)**

The methods, commands, that can be applied to widgets whose class is classname. It includes the methods that are inherited as well as the methods that are unique to this class. Every widget class created is automatically given the **config** or **configure** method which is given special treatment as described in the next section.

### **classname(optionName)**

optionName is a particular option specified in either of the root option or option lists. There must be a definition for each one of the options that is unique to this class. They should also have similar values to those of standard Tk option values,<sup>3</sup> with one exception. Since we are defining class options the objectValue should not be specified.

## **Widget Data Structure**

This is the second data structure that is created by the class builder and it is created for each widget whose class is classname. Thus, it defines the properties of a particular widget. The widget record is an array *r* whose name is the same as the widget's window path name. For example if *.p* is the path name for a widget then *.p(class)* would contain that widget's data structure. A useful technique is to store the window path names of the subwidgets<sup>4</sup> in the widget's data structure., this makes it easy to retrieve the path names later.

---

<sup>3</sup> Standard TK Options have one of two forms :  
{-abbreviation realOptionName} or {-optionName objectName className classValue objectValue}

<sup>4</sup> A subwidget is any widget that is contained in another widget. For example a scrollbar is composed of three subwidgets; a slider, top arrow and bottom arrow buttons.

**data(class)**

The class type the widget belongs to.

**data(className)**

The class name of the class type the widget belongs to.

**data(root)**

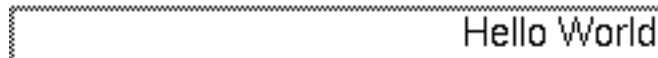
The window path name of the root widget. Due to how the widgets are created, this will always be a Tk frame widget; the widgets which are constructed in *classname\_ConstructWidget* are created inside of this frame.

**data(optionName)**

Defines the default object value for a particular option. If none is specified then the object option value is defined to be the class option value.

## Creating A Widget Class

There are a number of procedures that may or may not be defined in order to create a widget class. The GroupKit class builder gets its information from these procedures in order to create the new widget class. However, only *classname*, *classname\_ConstructWidget*, and *classname\_CreateClassRec* must be defined. An important thing to note is that the class does not have to be created before it is used; it will be created automatically the first time. Below is a diagram of the *myLabel* widget, which is used as an example throughout the discussion. *MyLabel* (implemented in the module *Label.tcl*) illustrates how default values can be given to a standard Tk label widget. Following the discussion there are two complete examples.



### **classname**

This procedure is called when a widget of the class *classname* is created. Typically it just calls "gkInt\_CreateWidget" with the appropriate arguments and returns the widget's path name as in the code segment below.

```
proc myLabel {w args} {
    eval gkInt_CreateWidget $w myLabel myLabelClass $args
    return $w
}
```

### **classname\_Config**

The configuration routine is activated whenever the *configure* command is invoked on a widget when at least one option/option value pair is specified. Therefore, his procedure specifies what is to be done when an option value is changed. This routine should have the widget's path name, option and option value as arguments. Note that the option and args parameters will each have a single value, not a list of values., thus, this routine will only operate on a single

option at any given time. Since the *myLabel* widget inherits all of its options from the Tk label widget then all that is required is to apply the option and its value to the actual label widget, as demonstrated below. If an error occurs in this routine and the option specified is valid, it is in one of the option lists, then an attempt will be made to apply the command to the root frame widget.

```
proc myLabel_Config {w option args} {
    upvar #0 $w data

    $data(label1) config $option $args
}
```

## classname\_ConstructWidget

This procedure constructs a widget for the class. Usually it consists of the creation and packing of a number of widgets. What the class builder does is take the widgets specified here and created them within a Tk *frame* widget, the root widget. It is also useful to store the each of the widget's path name in the widget's data record. The example below creates a label, stores the path name and packs it.

```
proc myLabel_ConstructWidget {w} {
    upvar #0 $w data

    set data(label1) [label $w.l1 -anchor $data(-anchor) \
        -background $data(-background) \
        -font $data(-font) \
        -foreground $data(-foreground) \
        -relief $data(-relief) \
        -text $data(-text) \
        -width $data(-width)]
    pack $data(label1) -fill x -expand t
}
```

## classname\_CreateClassRec

This procedure defines all of the class options, commands, and inheritance, i.e. all of the items discussed in the class builder overview section are defined here. Any of the undefined items are an empty list by default. The class record must be made a global variable so that widgets of this type can be constructed.

The code segment below creates the class record for *myLabel*. It inherits all of the commands and options from the Tk *label* widget and defines the *foreground* and *background* options.

```
proc myLabel_CreateClassRec {} {
    global myLabel

    set myLabel(inherit) {label}
    set myLabel(-foreground) {-foreground foreground \
        Foreground red}
    set myLabel(-background) {-background background \
        Background lightgray}
}
```

## classname\_DestroyWidget

Defines what is to be done when a widget of the particular class is destroyed.

## classname\_InitWidgetRec

Defines the default values for a widget's data structure. Typically all of the items described in the overview and often other variables that hold global information for the widget are defined in this procedure. The code segment below defines a number of default values for options.

```
proc myLabel_InitWidgetRec {w class className args} {
  upvar #0 $w data

  set data(-anchor)      e
  set data(-font)        \
    "-*-helvetica-medium-r-normal-*-14-*-*-*-*-*-*"
  set data(-relief)      raise
  set data(-text)        "Hello World"
  set data(-textvariable) textVar$w
  set data(-width)       30
}
```

## classname\_Methods

This procedure defines what action to take when any of the commands are invoked, with the exception of configure. Configure has some special properties, so it is treated slightly differently as described in the *classname\_Config* section above.

## classname\_SetWidgetBindings

Defines the various bindings for the class.

---

# Using A New Widget Class

As stated above, the intention was that a new widget class could be treated like existing widgets in Tk. For instance, the usage of new widget classes is no different than the usage of the *label* widget. The syntax for creating a widget of a particular class:

**class** *pathName ?option OptionValue ....?*

The example code below creates a *myLabel* widget.

```
myLabel .l
pack .l
```

The syntax for using widget commands is as follows:

**pathName command ?args?**

Below is an example of using the *config* command with the *myLabel* widget created above to change its *background* color:

```
.l config -bg Blue
```

---

## Known Problems

The biggest problem with this application is the manner in which the inherited options and commands get applied to the new class. It would be nice if these could automatically be applied to the new class in an appropriate manner, such as to all of the subwidgets. Perhaps in a later version this will be corrected. Currently it is left up to the designer of the new class to specify how the options and commands are used in the new class. As a result, errors can occur if this is not done.

---

## Examples

### Widget Class fancyLabel

This example demonstrates how special features can be added to an existing widget class, as well as inheritance. *FancyLabel* is essentially a *myLabel* widget, described above, with the additional option *reverse* and additional command *flash*. The *reverse* option allows the label to be displayed in reverse video, i.e. the background and foreground colors are switched. The *flash* command makes the label flash or blink.

#### Widget/Class Creation

The procedure below allows a new *fancyLabel* to be created given a particular window path name. Note that this is only place where any of the class builder routines are called, i.e. the *gkInt\_CreateWidget* command.

```
proc fancyLabel {w args} {
    eval gkInt_CreateWidget $w fancyLabel FancyLabel
    $args
    return $w
}
```

#### Class Record

The following procedure specifies that the *fancyLabel* inherits its options and commands from the *myLabel* widget. It also defines a new option, *reverse* and adds it to the option list. How the *reverse* option actually affects the widget is determined in the procedure *fancyLabel\_Config*. There is also a new command, *flash*. Again this is defined elsewhere in the procedure *fancyLabel\_Methods*. An important thing to note is that *fancyLabel* is declared to be a global value, this is to ensure that the *fancyLabel* widget class is known.

```
proc fancyLabel_CreateClassRec {} {
    global fancyLabel

    set fancyLabel(inherit) {myLabel}
    set fancyLabel(options) {-reverse}
    set fancyLabel(-reverse) \
        {-reverse reverse Reverse false}
```

```

        set fancyLabel(methods) {flash}
    }
}

```

## Widget Construction

The construction of the widget is done in the procedure below. This widget contains a *myLabel* widget only so it is created and its window path name is stored in the data structure for the widget. This is useful since other procedures may need to do operations on the subwidget. Finally, note that the `upvar` command is used to access the widget's data structure.

```

proc fancyLabel_ConstructWidget {w} {
    upvar #0 $w data
    set data(label1) [myLabel $w.l1]
    pack $data(label1) -fill x -expand t
}

```

## Changing Option Values

Next a routine is needed to interpret what is done when an option value is changed. If the option to be changed is *reverse* and the new value is different then the old value then the *foreground* and *background* colours of the widget are switched. An internal routine, *evaluateReverse*, determines if the new color is different from the old one. If the option is not *reverse* then it must be one of the valid options for the *myLabel* widget so the *myLabel* widget is configured with the given option and value.

```

proc fancyLabel_Config {w option args} {
    upvar #0 $w data

    if { [string match $option "-reverse"] } {
        if { [evaluateReverse $w $args] } {
            $w config -background $data(-foreground) \
                -foreground $data(-background)
        }
    } else {
        $data(label1) configure $option $args
    }
}

```

## Handling Commands

The next procedure handles the valid commands for the *fancyLabel* widget. If the command specified is *flash* then every 100 milliseconds the value of the *reverse* option is changed. An internal procedure, *evaluateReverse*, is used to determine what the option value should be changed too. Finally if the command specified is something other than *flash* it is assumed that the *myLabel* widget handles it so it is evaluated with it.

```

proc fancyLabel_Methods {w command args} {
    upvar #0 $w data

    if { [string match $command flash] } {
        if { [evaluateReverse $w true] } {
            $w config -reverse true
        } else {
            $w config -reverse false
        }
        after 100 "$w flash"
    } else
        eval $data(label1) $command $args
}

```

## Internal Procedures

Finally an internal procedure, below, returns true (1) if the new value, args, is different then the old value, \$data(-reverse). If the new value is not a valid value for the *reverse* option then an appropriate error message is given.

```
proc evaluateReverse {w args} {
    upvar #0 $w data

    set flag 0
    switch -regexp $args {
        t|true|1 {
            if { [regexp -nocase fa*|0 $data(-reverse)] }
            {
                set flag 1 }}
        f|false|0 {
            if { [regexp -nocase tr*|1 $data(-reverse)] }
            {
                set flag 1 }}
        default {error "bad option value \"$args\": must \
            be a boolean value"}
    }
    return $flag
}
```

## Using the FancyLabel Widget Class

Since all of the necessary procedures are defined above for the class fancyLabel, a widget of this class can be created the same manner as any Tk widget. Since *reverse* is a valid option, it may be part of the option list for the widget. The syntax is as follows:

```
fancyLabel pathName ?option optionValue ... ?
pack pathName ?option optionValue ... ?
```

Where pathName is the window path name and option/optionValue pairs are any valid option specification for the *fancyLabel* widget. Below is an example which creates a fancyLabel widget.

```
fancyLabel .l -reverse false
pack .l
```

Once again the intention was for widgets created by the class builder is to act like existing widgets in Tk, so using a command on this widget is the same as any other Tk widget as follows:

```
pathName command ?args?
```

Where pathName is the window path name for a *fancyLabel* widget and the command is a valid command. Continuing the example from above the code segment below will make the widget *flash* then get the value of the *reverse* option.

```
.l flash
.l config -reverse

# Result of the "config" command above.
-reverse reverse Reverse false false
```

Finally a picture of the *fancyLabel* widget.



## Creating Widget Class *frameLabel*

This example describes the composite widgets, *frameLabel* is composed of a *fancyLabel* and a *frame* widgets. *FancyLabel* is described above and is created with the GroupKit class builder and *frame* is a standard Tk widget.

### Widget/Class Creation

The procedure below allows a new *frameLabel* to be created given a particular window path name. This is only place where any of the class builder routines are called, i.e. the *gkInt\_CreateWidget* command.

```
proc frameLabel {w args} {
    eval gkInt_CreateWidget $w frameLabel FrameLabel
    $args
    return $w
}
```

### Class Record

The following procedure specifies that the *fancyLabel* class record. It inherits all of the commands, options and option values from *fancyLabel* as well as *frame*; this includes the command *flash*. If an option is known to both the *fancyLabel* and *frame* classes then the value used will be the value defined in the *fancyLabel* class because it has a number of new options which manipulate the *frame* widget's option values. These new options are as follows:

```
-framebd      border width of the frame
-framebg      background color of the frame
-framegeom    geometry of the frame
-framerelief  relief of the frame
```

The class record procedure is below.

```
proc frameLabel_CreateClassRec {} {
    global frameLabel

    set frameLabel(inherit) {fancyLabel frame}
    set frameLabel(options) {-framebd -framebg -framegeom \
        -framerelief}
    set frameLabel(-framebd) {-framebd frameBd FrameBd 3}
    set frameLabel(-framebg) {-framebg frameBg FrameBg \
        gray50}
    set frameLabel(-framegeom) {-framegeom frameGeom \
        FrameGeom 30x30}
    set frameLabel(-framerelief) {-framerelief frameRelief \
        FrameRelief ridge}
}
```

### Widget Construction

Next the widget is constructed in the procedure below. A *fancyLabel* and a *frame* widget are created and their window path names are stored in the widget's

data structure. This is useful since other procedures may need to do operations on the widget. The `upvar` command is used to access the widget's data structure.

```
proc frameLabel_ConstructWidget {w} {
    upvar #0 $w data

    set data(frame1) [frame $w.fl \
        -borderwidth $data(-framebd) \
        -bg $data(-framebg) \
        -relief $data(-framerelief) \
        -geometry $data(-framegeom) ]
    set data(label1) [fancyLabel $w.l1]
    pack $data(frame1) -fill both -expand t
    pack $data(label1) -fill x -expand t -padx 10
        -pady 10 -anchor center -in $data(frame1)
}
```

### Changing Option Values

Next a routine that handles changes in a widget's option values. If the option to be changed is a frame option, starts with “-frame”, then apply the option and option value to the *frame* widget. If the option is not a frame option then apply it to the *fancyLabel* widget.

```
proc frameLabel_Config {w option args} {
    upvar #0 $w data

    if { [string match *frame* $option] } {
        set option [string range $option 6 end]
        $data(frame1) config -$option $args
    } else {
        $data(label1) config $option $args
    }
}
```

### Handling Commands

The next procedure handles the valid commands that can be used with the *fancyLabel* widget. Since all of the commands are inherited from the *fancyLabel* then the command is applied directly to the *fancyLabel* widget. Since some of the commands also apply to the frame, but not all, `catch` is used so that only the appropriate commands are issued on the *frame* widget.

```
proc frameLabel_Methods {w command args} {
    upvar #0 $w data

    $data(label1) $command $args
    catch {$data(frame1) $command $args}
}
```

## Using the FrameLabel Widget Class

A widget of this class can be created the same way as any widget is created in Tk. *Reverse* is a valid option, so it may be part of the option list for the widget. The syntax is as follows:

```
fancyLabel pathName ?option optionValue ... ?
pack pathName ?option optionValue ... ?
```

Where `pathName` is the window path name of a widget and the `option/optionValue` pairs are any valid option specification for the `myLabel` widget. Below is an example which creates a `fancyLabel` widget.

```
frameLabel .l
pack .l
```

Once again the intention was for widgets created by the class builder is to act like existing widgets in Tk, so using a command on this widget is the same as any other Tk widget as follows:

**`pathName command ?args?`**

Where `pathName` is the window path name for a `fancyLabel` widget and the `command` is a valid command. Continuing the example from above the code segment below will make the widget flash then get the value of the `reverse` option then get the value of the frames background color.

```
.l flash
.l config -reverse
.l config -framebg
```

```
# Result of the "config" commands above.
-reverse reverse Reverse false false
-framebg frameBg FrameBg gray50 gray50
```

Finally a picture of the `frameLabel` widget.



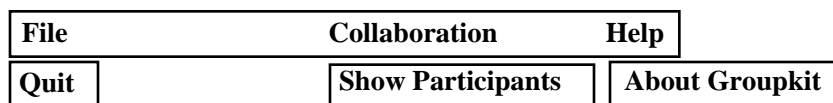
# GroupKit Widgets

This chapter describes the set of widgets that are provided by GroupKit for use in your applications. These widgets consist of several widgets useful in any application, such as an about box, and some that are particular to groupware applications, such as shared telepointers. The source code for the widgets resides in the *widget* directory, the GroupKit class builder is used in the implementation of these widgets.

---

## GroupKit Menu Bar

GroupKit supplies a default menu bar that should be displayed by all GroupKit applications. The menu bar is minimally configured with three pulldown menus; *File*, *Collaboration* and *Help* menus. It looks something like:



*Quit* allows graceful exits.

*About GroupKit* gives background information on GroupKit.

*Show Participants* displays who is in the conference.

One routine is provided to create the menubar widget (implemented in the module *gk\_widget\_menubar.tcl*).

```
gk_defaultMenu pathName ?options?
```

Create a menubar with the path *pathName* containing the file, collaboration and help pulldown menus. The options that can be specified are the same as the ones provided for the *menuButton* widget. The option values can be changed with the *configure* command.

The indices used to denote the various menus of the menubar can be specified in two different ways; a character string or an integer value. The string is the name of the menu while the integer is the position of the menu on the menubar. The leftmost menu has index 0. *Note that the string values are not guaranteed to be unique so for reliable results use the integer indices.*

Programmers can add and delete menus from the menubar, as well as, get a list of the current menus on the menubar. Below are a couple of examples.

```
# Add a conference menu  
.menubar add confmenu 1 -text Conferences
```

```

# Remove the Collaboration menu from the default menubar
.menubar delete collaboration
# Display the menubar list
.menubar itemList

```

Programmers can also perform operations on the various menus themselves. The *itemcommand* command allows any menu command to be executed on a particular menu and the *itemconfigure* command allows any of the option values of a menu to be changed. The following code segment from the *brainstorm.tcl* file adds a number of items to the *file* menu.

```

.menubar itemcommand 0 add separator
.menubar itemcommand 0 add command -label "Open" \
    -command "Open"
.menubar itemcommand 0 add command -label "Save" \
    -command "Save"
.menubar itemcommand 0 add separator
.menubar itemcommand 0 add command -label "Clear Ideas" \
    -command "Clear"

```

## Participant Window

GroupKit supplies a window which displays all of the participants in the current conference application. The window is automatically updated as people enter and leave the conference. Depending on the value of the *show* option, the names of participants are either labels or radio buttons. If they are radio buttons, selecting a participant will bring up an *attribute* window described in the next section.

Here is what it looks like, showing three people participating in the conference application “Minimalist Brainstormer”.



This widget is called by the *Show Participants* item in the GroupKit default menubar. Programmers can call it directly (implemented in the module *gk\_widget\_participants.tcl*).

```
gk_participants pathName ?options?
```

Creates a window that displays a continually updated list of all participants in the current conference application. The option values can be changed with the `configure` command. This widget inherits all of the options and commands from the Tk toplevel widget, it also has some others. However, the only option of importance is the `show` option. If this option is true then the display contains radio buttons ;if it is false then labels are used. The default value for this option is “true”.

## Attribute Window

This window shows all the currently known attributes of a user (as stored in the users environment). This is currently useful for debugging; in a future release it will be modified to provide information that is truly useful to the end user e.g., contact information.

This widget is called by the radiobuttons in the *participant window*.. Programmers can call it directly (implemented in the module *gk\_widget\_attributes.tcl*)

```
gk_attributes pathName ?options?  
e.g.. gk_attributes .attrib -usernum 3
```

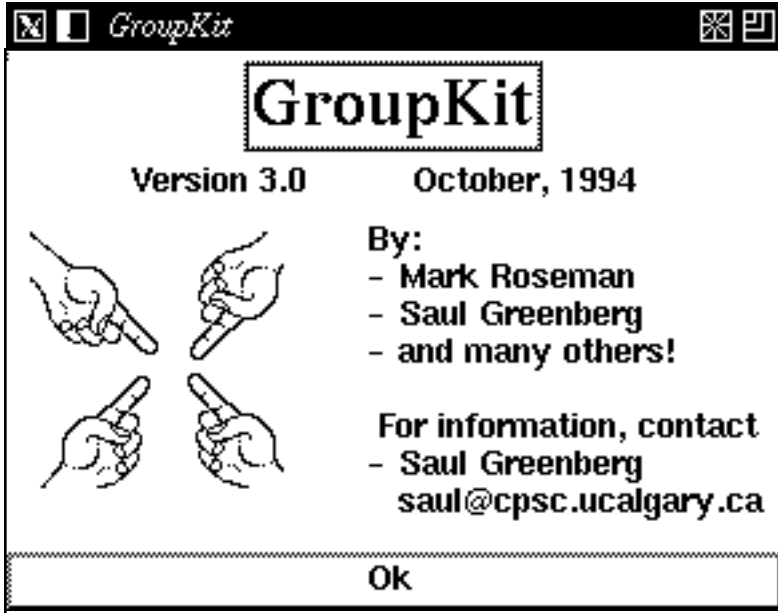
Creates a window that display a users attributes for the user specified by the *usernum* option. If the *usernum* option is not specified then no attributes are displayed. The *configure* command can be used to change the value of any of the options.

A programmer can also add and delete attributes from the window. The code segment below add the attribute *active* and deletes the attribute *usernum*. in the attribute window “.attrib”.

```
.attrib add active true  
.attrib delete usernum
```

## About Box

GroupKit supplies an information window that describes itself i.e., version number, contact information, etc. It currently looks like:



This widget is called by the *About GroupKit* item in the GroupKit default menubar. Programmers can call it directly (implemented in the module *gk\_widget\_about.tcl*).

```
gk_about pathName ?options?
```

Creates a window that displays some information about GroupKit. This widget is composed of several parts which can all be changed by changing the option values for the widget; this can be done with the *configure* command. For instance the *bitmap* option specifies the bitmap that is displayed and the *version* option changes the version number.

## Help

GroupKit application should provide the user with a brief description of the application and how to use it. The user would normally select it through the "Help" pulldown menu in the GroupKit menu bar.

The topic window widget, described in the next section, allows the help information to be formatted and appears in a scrollable window. A typical example of a help information box and the code that created it.



```
.menu itemcommand help add command \
-label "$help_title" \
-command "gk_topicWindow .helpWindow \
-title {$help_title} \
-text {$help_text}"
```

*menu* is the default GroupKit menu bar that should have been created before.

*help\_title* is a short descriptive phrase describing the help description.

*help\_text* is formatted help text..

This will add the help label to the help menu and tells what command, i.e. what should be displayed, when this menu item is selected. For more information on the format of the help description see the section on the *topicWindow*.

## Topic Window

This widget will display some text in a given format, it is typically used for displaying help information but need not to be. Construct a topic window (implemented in *gk\_widget\_topicWindow.tcl*).

```
gk_topicWindow pathName ?options?
```

The *title* option specifies the title of the message and the *text* option specifies the message itself. Both of these should be in the format described below.

```
{ {font type} {text}
  {font type} {text} ...
}
```

The font type is one of:

large	normal
largebold	normalbold
largeitalic	normalitalic
largebolditalic	normalbolditalic

### Example.

The following excerpt defines some help text which will be used as help information and also creates the *topicWindow*. See the section on help to see how it can be added to the default menubar, as well as a diagram of the output. Notice that the formatted syntax of the text is not nice but it does work.

```

# Define the formatted text
set help_title "About Simple Sketchpad"
set help_text {
  {normal} {This is a very simple groupware sketchpad.

  }{normalbold} {Press } {normal} {and } {normalbold} {drag
  }
  {normalitalic} {mouse button 1 }
  {normal} {to draw on the displays of \
all participants.}
}

# Create the window
gk_topicWindow .helpWindow \
  -title {$help_title} \
  -text {$help_text}

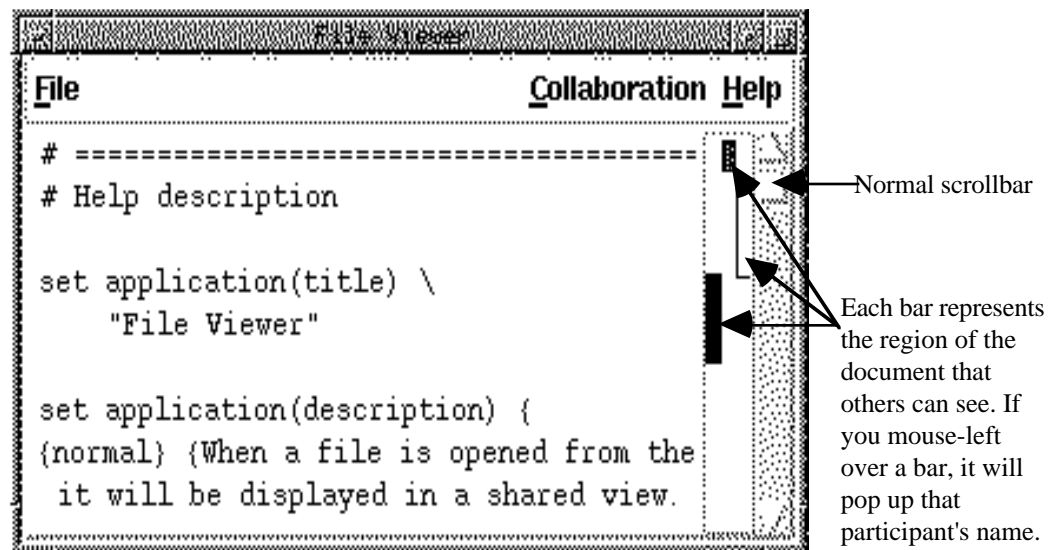
```

---

## Multi-User Scrollbar

*Warning: The GroupKit Scroll Bar is under development and will probably change in future releases. We are also aware of a small non-critical bug, where parts of the scrollbar will not update immediately when a new conference participant arrives. This usually fixes itself as soon as the new person scrolls.*

GroupKit supplies a special scrollbar that not only shows your location in a view, but other people's as well. It is comprised of a standard TK scrollbar (on the right), and a special region on its left that shows a bar representing where others are and the relative size of the region they can see. If you mouse-left over a bar, it pops up the participant's name. It looks something like this:



One routine is provided to construct the scrollbar (implemented in the module `gk_widget_scrollbars.tcl`).

**gk\_groupScroll** *pathName ?option OptionValue ... ?*

Create a groupkit scrollbar with the widget path *pathName*. As with normal scrollbars, you should attach it to whatever other widget you want to scroll. Note it only implements a vertical scrollbar. This will change in future releases.

The group scrollbar acts much like a standard Tk widget. Its options and commands are inherited from the Tk scrollbar widget plus it has a few of its own described below. Like any other widget the configure command can be used to change options as well as evaluate them.

### Additional Options

**Barwidth** specifies the width of the multi-user scrollbar. This does not include the width of the manipulable scrollbar. Any value that is acceptable by the width option for the canvas widget may be used. If this option is not specified then it defaults to 20.0 pixels.

**Color** specifies the color of the "local" user's scrollbar in the multi-user portion of the scrollbar. Any of the color values used by Tcl/Tk are acceptable. If this value is not specified then the `gk_getMyColour` command is used to determine the user's color.

### Commands

**pathName local** returns the window pathName of the real scrollbar on the right.

**pathName remote** returns the window pathName of the multi-user scrollbar on the left.

**gk\_scroller pathName args** is used to tie into the `-yscrollcommand` of whatever widget is being scrolled. It expects the same arguments as a normal scrollbar set command. See the example below.

### Example.

The following code is a complete GroupKit application that creates a group scrollbar, a listbox with some elements, and joins them together. Type it in at try it out!

```
gk_initConf $argv

listbox .lbox -yscrollcommand "gk_scroller .vscroll"
gk_groupScroll .vscroll -command ".lbox yview"

pack .vscroll -side right -fill y
pack .lbox -side left -expand yes -fill both

foreach item {a b c d e f g h i j k l m n o p q} {
    .lbox insert end $item
}
```

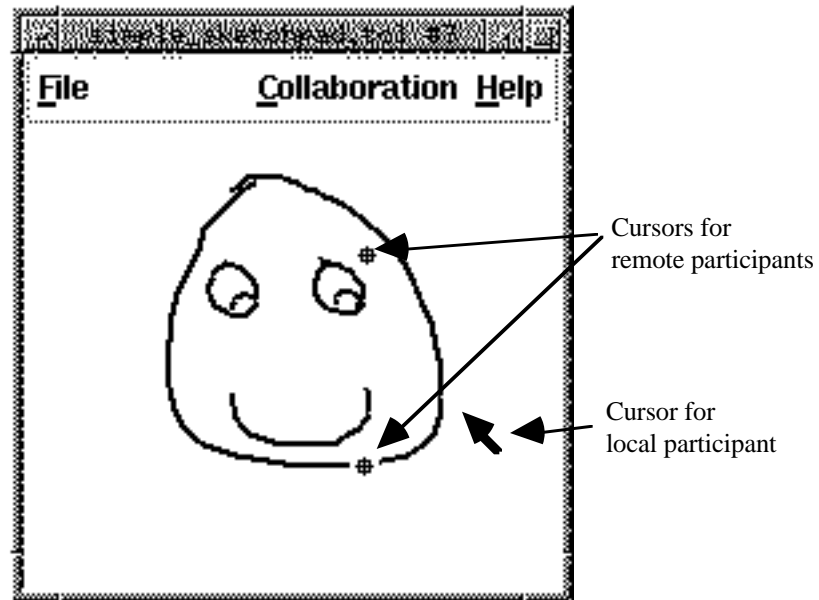
---

## Telepointers

*Warning: The GroupKit Telepointer is under development and will probably change in future releases.*

GroupKit offers the ability to add telepointers, or multiple cursors, to its applications on a widget by widget basis. Telepointers currently track relative to a widget, if widgets appear in different locations on different participants' screens, the telepointers will appear in the correct place. Currently, telepointers are not smart enough to appear in the correct relative position within a scrollable widget (this will change in future releases).

For example, here is the simple sketchpad showing three people using it, showing the local cursor and two telepointers.



Several routines are provided to activate and manipulate telepointers.

#### **gk\_initializeTelepointers**

Initialize our telepointer, which also tells other participants about it. However, this doesn't actually attach it to anything.

#### **gk\_specializeWidgetTreeTelepointer *widget***

Display the telepointer in the widget specified by *widget* and all of its children. The telepointer will appear relative to the widget, even if they are on different locations on other displays. In any single GroupKit application, you can thus selectively add telepointing capabilities to all, some or none of the widgets. For example, you can call this routine twice to add telepointers to (say) a canvas and a button, while ignoring the menubar.

#### **Example.**

The following complete program adds telepointers to an otherwise empty GroupKit conference.

```
gk_initConf $argv
```

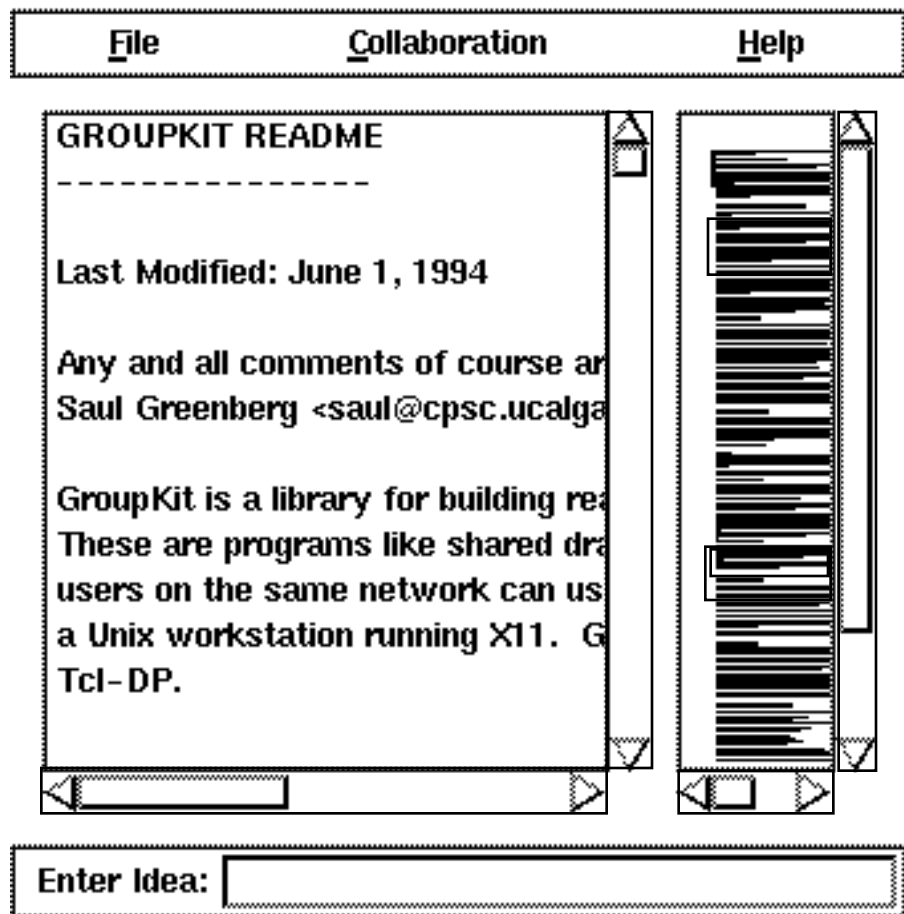
```
gk_initializeTelepointers
gk_specializeWidgetTreeTelepointer .
```

---

## Gestalt View

*Warning the Gestalt View is currently under construction and may change in future releases.*

Gestalt view is a method of relaying the location of users. Each user has a colored box representing the section of a particular application that they are currently viewing. It is assumed that each object of the application is represented by a single line in the gestalt view; these lines have varying lengths, but not widths. The length of each line corresponds to the size of the object it represents. It is assumed that this widget is used with a GroupKit application and that the conference has already been initialized, i.e. `gk_initConf` has been called. Finally, this application is built on top of `gkClassBuilder` and the `gkViewport`. These allow the gestalt view to be treated like a single widget and add scrolling features respectively. The module `gestaltViewBrainstorm.tcl` uses the gestalt view. Below is a picture of it.



## Functionality

The gestalt view is a Tk canvas widget with some fancy features. Thus, any of the options and commands that can be used with the Tk canvas widget can be used with the gestalt view; the additional commands and options are listed below. The second mouse button scrolls the gestalt view in the same manner as the Tk listbox widget, and the first mouse button is associated with the views. A single click allows "dragging" of the local view only if the drag option is set to true. A double click will display a list of owners of all the boxes that are close to the mouse.

A gestalt view (implemented in the *gk\_widget\_gestaltView.tcl*) can be created with the following command.

```
gk_gestaltView pathName ?options?
```

### Options

As stated above, the gestalt view has all of the options that the canvas widget has, however, it does have a few extra that are described below. They can be changed with the configure command.

The **canvasborder** is the border between the lines and the edge of the canvas.

**Command** is the command executed when there is a change in a box's position, probably due to it being dragged. Whose box, as well as the top and bottom of the box, are appended to the command.

**drag** either enables or disables the dragging of the local box.

**scroll** specifies where the scroll bars should be located. If there is no value given to this option then no scrollbars are shown. Valid values are left, right, top and bottom.

### Commands

The gestalt view also has a number of commands, besides the ones inherited from the Tk canvas widget, that make using it easy as well as offering lots of information. These commands are described below.

**pathName boxcoords id** will return the top and bottom values of the local user's box.

**pathName boxexists id** returns true if a user, *id*, has a box in the gestalt view and false if it does not.

**pathName createbox id top bottom color** creates a new box for a particular user. It requires the user's Id, the top and bottom of the box as well as the color to be used. The user's Id should be a unique number that identifies the user. Usually the GroupKit "usernum" is used.

**pathName delete first ?last?** deletes the lines, inclusively, from "first" to "last". Both first and last should be index values, i.e. integers or "end". If the last index is not given then only the line given by the first index is removed.

**pathName deletebox id** removes the box of a particular user. Where Id is the user's unique identification, usually the "usernum" assigned to the user by GroupKit.

**pathName end** returns the index of the last line. Note that the first index is always 0.

**pathName insert index length** insert one line at a given index. The index and length are integer values.

**pathName movebox window id top ?bottom?**

Moves a box to a specified position. Window is the path name of the window causing the box to be moved. This is used to determine whether or not the value of the -command option should be executed. Id is the user's unique identification, usually the "usernum" assigned to them by GroupKit. Top and bottom are the top and bottom positions of the box respectively and can be valid index. If the bottom value is not specified it is assumed that the boxes size has not changed.

## Viewport

This widget attaches a vertical and a horizontal scrollbar to another widget, the child. The position of the scrollbars depends on the value of the scroll option. If the xscrollcommand option is not defined for the child widget then it is given a default value as is the "command" option for the vertical scrollbar. The vertical scrollbar is given default values similarly. *If the child widget does not have a xscrollcommand or a yscrollcommand an error will occur.* The TK scrollbar allows the view be dragged to a point where there is only one item at the top of the screen, whereas in this design the view is dragged until the last item in the listbox is at the bottom (rather than the top) or the view. A viewport widget (implemented in the *gk\_widget\_viewport.tcl*) can be created with the following command.

```
gk_viewport pathName child ?options?
```

### Options

The only option which is not inherited from the Tk scrollbar widget is the scroll option. It specifies which scrollbars are to be attached. The value it accepts is a list consisting of 0 or more of the following: left, right, top and bottom. Only one vertical and one horizontal scrollbar are permitted. Both of left/right or top/bottom can not be specified, however, if they are right and bottom take precedence. If no value is specified then the default value is: {right bottom}.

### Commands

The viewport widget inherits all of the commands from the scrollbar widget plus the following commands:

**pathName repack** causes the viewport to be repacked.

**pathName xcommand** returns the xscrollbar command for the horizontal scrollbar.

**pathName xscrollbar** returns the window name of the horizontal scrollbar.

**pathName ycommand** returns the yscrollbar command for the vertical scrollbar.

**pathName yscrollbar** returns the window name of the vertical scrollbar.

# Conference Applications

This section of the paper goes through a few complete conference examples. All of these examples have been included in the distribution of GroupKit. You may also want to look at the Hello World chapter which discusses a very simple conference application. The brainstorming application is a more complicated one that describes what a typical groupware application might be. Finally, the drawing program illustrates how using environments can be used to share information.

---

## Example: Brainstorming

This section walks through a complete example of a GroupKit conference, the anonymous brainstorming tool. It illustrates how to structure a conference application, and provides examples of sending messages, handling GroupKit-generated events, and using some of the GroupKit widgets.

### Initializing the Conference Application

The first thing any GroupKit conference application must do is initialize GroupKit. This takes care of a number of details, such as connecting back to the Registrar Client that spawned the conference (to receive various registration events, such as users joining and leaving), and initializes a number of important data structures, such as the “users” environment.

The initialization is performed by GroupKit’s *gk\_initConf* routine, which takes as a parameter the command line arguments used when the conference application process was created. These arguments were provided by the Registrar Client, and provide important information about the conference, such as the conference number, its name, and the local user number.

```
gk_initConf $argv
```

Note that the *gk\_initConf* procedure is located in *conf.tcl*. Calling the routine automatically loads *conf.tcl* into memory, via the standard Tcl autoload mechanism.

### Building the Menus

Next, we begin to build the user interface, starting with the menu bar. We’ll use the menu bar supplied by GroupKit (see the “GroupKit Widgets” section). We start with the basic menu bar:

```
gk_defaultMenu .menubar
```

We want to provide on-line help for our application. First we need to define both the name of our application and some help text which describes it:

```
set application(title) "Brainstormer"

set application(description) {
{normal} { Type an idea into the ideas area and hit
return.

The idea will display in the shared ideas list, which is
scrolled
  by the multi-user scroll bar, or by pressing the }
{normalitalic} {middle mouse button.

}
{normal} {The }
{normalitalic} {File }
{normal} {menu will let you save ideas to a file, open\
an existing file, or clear the ideas from the display

}
{normalbold} {Limitations:
}
}
{normal} {Control characters are handled poorly.}}
```

We then add a help item to the menu:

```
.menubar itemcommand help add command \
-label "$application(title)" \
-command "gk_topicWindow .helpWindow \
-height 12 \
-title {$application(title)} \
-text {$application(description)}"
```

We continue to build up the menu bar. We want to supplement the items in the existing GroupKit File menu, so we get that via the *gk\_menu* variable.

```
.menubar itemcommand 0 add separator
.menubar itemcommand 0 add command -label "Open" \
-command "Open"
.menubar itemcommand 0 add command -label "Save" \
-command "Save"
.menubar itemcommand 0 add separator
.menubar itemcommand 0 add command -label "Clear Ideas"
-command "Clear"
pack .menubar -side top -fill x
```

## Building the Main Interface

We begin with a few standard Tk commands to set up the toplevel window properly, i.e. so that it can be resized and has an appropriate title.

```
wm title . $application(title)
wm minsize . 20 7
```

We then add a frame which will take up most of the window, and place within it the listbox which will holds the ideas users generate.

```
listbox .middle.shared_list \
-setgrid true
```

```

gk_viewport .middle.v .middle.shared_list -scroll {}
set ycommand [.middle.shared_list config -yscrollcommand]
.middle.shared_list config \
    -yscrollcommand "[list scroller [lindex $ycommand 4]]"

```

Normally a listbox will have a scrollbar. The idea list does as well, except rather than a normal scrollbar, we'll use GroupKit's multi-user scrollbar. The syntax is similar to that used when adding a normal scrollbar.

```

gk_groupScroll .middle.vscroll \
    -command ".middle.shared_list yview"

```

We next pack the listbox and scrollbar together:

```

pack .middle -side top -expand yes -fill both
pack .middle.vscroll -side right -fill y
pack .middle.v -side left -expand yes -fill both

```

We then go on to define the rest of the interface, which includes a text entry field at the bottom used to initially enter ideas. When the *return* key is pressed, the procedure *insert\_idea\_into\_shared\_list* (discussed below) is called.

```

frame .bottom
label .bottom.label -text " Enter Idea:"
entry .bottom.idea_entry -textvariable myIdea \
    -relief sunken
focus .bottom.idea_entry

pack .bottom -side bottom -fill x
pack .bottom.label -side left
pack .bottom.idea_entry -side left -expand yes -fill x

bind .bottom.idea_entry <Return>\
    insert_idea_into_shared_list

```

The next little procedure updates the groupscrollbar and the viewport widget. Note that the viewport widget makes sure that the last line of the entire document never gets scrolled up to the top of the listbox, it stays at the bottom.

```

proc scroller {viewCommand args} {
    eval gkGroupScroll_scroller .middle.vscroll $args
    eval $viewCommand $args
    update
}

```

## Broadcasting Changes

We now get to the place where newly entered ideas are sent to other conference users. We use the *gk\_toAll* command, which calls the procedure *actually\_insert\_idea\_into\_shared\_list* for each user, including the local user who initiated the idea.

```

proc insert_idea_into_shared_list {} { global myIdea
    if {$myIdea != ""} {
        gk_toAllactually_insert_idea_into_shared_list\
            $myIdea
        set myIdea ""
    }
}

```

The next procedure is the one that actually inserts the idea into the listbox.

```

# Actually insert the idea into the shared list
proc actually_insert_idea_into_shared_list {idea} {
    .middle.shared_list insert end $idea
}

```

This sort of construct is used quite frequently in GroupKit programs that you write — a pair of procedures, where the first one, invoked from the interface, merely calls `gk_toAll`, leaving the real work to the second procedure.

## Updating New Entrants

The next thing we want to do is update newcomers to the conference. In this case, when a new user joins, we want one user in the conference to send them a current copy of the list of ideas that have been generated. We'll use the `updateEntrant` event to do this.

The first step is to set up a trigger to catch the `updateEntrant` event:

```

gk_on {[gk_event type]=="updateEntrant"} {
    updateEntrant [gk_event usernum]
}

```

When it does occur, we send our list of ideas to the new user. Effectively, this is replaying all the “`gk_toAll`” calls that have inserted ideas into the list to date.

```

proc updateEntrant {usernum} {
    for {set count 0} {$count<[.middle.shared_list size]}\
        {incr count 1} {
        set idea [.middle.shared_list get $count]
        gk_toUserNum [gk_event usernum] \
            actually_insert_idea_into_shared_list $idea
    }
}

```

## Odds and Ends

The rest of the program deals with implementing the commands we added to the File menu: Save, Open and Clear. While Save is relatively straightforward, the others beg some interesting questions about the software design. Should any user be able to delete all the existing ideas in the list? Here we take a “social protocol” approach, assuming the group is well enough behaved and can negotiate such potentially dangerous actions.

```

proc Save {} {
    if {[.middle.shared_list size] == 0} {return}
    set fileName [FSBox]
    if {$fileName == ""} {return}
    set fd [open $fileName a]
    for {set count 0} \
        {$count < [.middle.shared_list size]} \
        {incr count 1} {
        puts $fd [.middle.shared_list get $count]
    }
    close $fd
}

proc Open {} {
    set fileName [FSBox]
}

```

```

    if {$fileName == ""} {return}
    Clear
    set fd [open $fileName r]
    set result [gets $fd temp]
    while {$result != -1} {
        gk_toAllactually_insert_idea_into_shared_list\
            $temp
        set result [gets $fd temp]
    }
    close $fd
}

proc Clear {} {
    gk_toAll doClear
}

proc doClear {} {
    .middle.shared_list delete 0 end
}

```

---

## Example: Drawing Program

This section walks through program that allows users to draw and manipulate ovals on a canvas widget. While the brainstorming example relied on explicit message passing to share state — the list of ideas — with other users, the drawing program relies on shared environments. The result is that this multi-user drawing program is scarcely different from a single user version, albeit one designed around providing a view of an underlying data structure of drawing objects. This example is a trimmed down version of the Hypertext Browser in the distribution.

First initialize GroupKit and create the overall interface. Here we'll use GroupKit's provided menubar, and also create a canvas that will contain all the nodes that we'll draw.

```

gk_initConf $argv
gk_defaultMenu .menubar
canvas .c
pack append . .menubar {top fillx} .c top

```

Next create two environments. The first, called “graph” is a shared environment holding information about the underlying representation of our objects. In this program its pretty simple — for each of the ovals, all we need to keep track of is its coordinates. The environment is shared so that if any of the users changes the coordinates of an oval — or adds or deletes an oval — all other users will know about it.

There is also have a second environment, “graphinfo”, which is not shared and keeps track of information of interest only to this process. We'll use this environment to associate each of the ovals stored in the shared “graph” environment with an onscreen representation — in this case, the id number of an oval object in the canvas widget. The idea will be that when the underlying object in “graph” changes, we can find the corresponding canvas object in “graphinfo” and update it accordingly.

```

gk_newenv -notify -share graph
gk_newenv graphinfo

```

Each oval will be referred to by a unique id number. The nextId procedure will generate a new id number; it guarantees that it will be unique by prefixing a counter with the local user's user number and an "x", e.g. "35x1".

```
graphinfo misc.nextid 0

proc nextId {} {
    set i [graphinfo misc.nextid]
    set id [users local.username]x$i
    incr i
    graphinfo misc.nextid $i
    return $id
}
```

The telepointers provided by GroupKit are used as well. These two calls take care of providing the telepointers atop the canvas widget we created.

```
gk_initializeTelepointers
gk_specializeWidgetTreeTelepointer .c
```

Now we'll start on the code that will initially create the ovals. We set a binding on the canvas widget so that when the left mouse button is first pressed it will call our "startNode" procedure, and as the user drags out the initial oval it will call "continueNode".

```
bind .c <1> {startNode %x %y}
bind .c <B1-Motion> {continueNode %x %y}
```

When the oval is first started, we generate a new id number for it, which will uniquely identify it in the conference. We save the starting position to "anchor" one of the oval's corners while drawing it out. Finally, we add information about the node — in this case, just its coordinates — to the shared "graph" environment. This will alert all users (via an "addEnvInfo" event, see below) that a new oval has been created.

```
proc startNode {x y} {
    graphinfo misc.theid [nextId]
    graphinfo misc.startx $x
    graphinfo misc.starty $y
    graph nodes.[graphinfo misc.theid].coords \
        [list $x $y $x $y]
}
```

When the node is initially dragged out, this procedure is repeatedly called. All it does is change the coordinates of the node we previously placed in the environment, reflecting its new size. The change will similarly be made available to all (via a "changeEnvInfo" event, see below).

```
proc continueNode {x y} {
    graph nodes.[graphinfo misc.theid].coords \
        [list [graphinfo misc.startx] \
            [graphinfo misc.starty] $x $y]
}
```

So far all that has been done is information in the environment is changed; we actually have not displayed anything on screen. So two triggers are set-up that will watch the environment and reflect any changes in the interface. The first trigger, for an "addEnvInfo" event will cause a new oval to be created on the canvas, while the "changeEnvInfo" trigger will react to a change of coordinates by moving an oval already on the canvas.

```

gk_on {[gk_event type]=="addEnvInfo"} && \
      {[gk_event env]=="graph"} {
  set id [parseNodeId [gk_event key]]
  if {$id!=""} {newNode $id}
}

gk_on {[gk_event type]=="changeEnvInfo"} && \
      {[gk_event env]=="graph"} {
  set id [parseNodeId [gk_event key]]
  if {$id!=""} {moveNode $id}
}

```

Next is a utility procedure used by the triggers described above. It will take a key (e.g. "nodes.35x1.coords") and return the id number of the node that the key describes (e.g. "35x1").

```

proc parseNodeId key {
  set pieces [split $key .]
  if {[lindex $pieces 0]=="nodes"} {
    return [lindex $pieces 1]
  }
  return ""
}

```

Here an oval is created in the canvas that corresponds to the underlying object "id" in the "graph" environment. After creating the oval, we store its canvas id in the "graphinfo" environment, so that given the object id from "graph" we can get at the corresponding canvas object. We also set up bindings on the new object that allow it to be dragged around after it has been created.

```

proc newNode id {
  scan [graph nodes.$id.coords] "%d %d %d %d" \
      x0 y0 x1 y1
  set cid [.c create oval $x0 $y0 $x1 $y1]
  .c bind $cid <3> "startDrag $id %x %y"
  .c bind $cid <B3-Motion> "continueDrag $id %x %y"
  graphinfo nodes.$id.canvasid $cid
}

```

This routine responds to changes in the coordinates of an underlying object in "graph". It finds the corresponding object — via "graphinfo" — and changes its coordinates to match the underlying object. The "update idletasks" is a performance tweak to ensure the ovals move smoothly; without it, the screen will not be updated until all other events have been processed.

```

proc moveNode id {
  scan [graph nodes.$id.coords] "%d %d %d %d" x0 y0 x1 y1
  .c coords [graphinfo nodes.$id.coords] $x0 $y0 $x1 $y1
  update idletasks
}

```

The two routines below allow a user to pick up an oval and drag it after it has already been created. Again, they modify the underlying object's coordinates stored in the "graph" environment, letting the events generated by the environment take care of updating the corresponding canvas object.

```

proc startDrag {id x y} {
  graphinfo misc.lastx $x
  graphinfo misc.lasty $y
}

proc continueDrag {id x y} {

```

```
set dX [expr $x-[graphinfo misc.lastx]]
set dY [expr $y-[graphinfo misc.lasty]]
graphinfo misc.lastx $x
graphinfo misc.lasty $y
scan [graph nodes.$id.coords] "%d %d %d %d" \
    x0 y0 x1 y1
incr x0 $dX; incr y0 $dY; incr x1 $dX; incr y1 $dY
graph nodes.$id.coords [list $x0 $y0 $x1 $y1]
}
```

# Registration

Not only can new conference applications be built in GroupKit, but new registration interfaces to the conferences can also be built. This section describes GroupKit's registration infrastructure and how it can be used to build new Registrar Clients. But beware: Registrar Clients can be considerably more difficult to construct than conferences!

This section does not go into great detail on the overall motivation for building different types of registration systems, or the philosophy behind the open protocols technique that GroupKit's registration system relies on. For more information on these topics, see the *Open Protocols* paper referenced in the Introduction to this manual.

There are three examples registrar clients included in the distribution of GroupKit 3.0; *open.reg.tcl*, *rooms.reg.tcl* and *facil.reg.tcl* all are registrar clients. The example presented in this section is *open.reg.tcl*.

---

## Overview of the Registration Architecture

GroupKit's registration architecture consists of two pieces, the central Registrar and the Registrar Clients. While the Registrar serves as the master repository for all conferences and their users, the Registrar Clients decide how to use that repository, initiating and reacting to changes in order to define a registration policy.

### The Role of the Registrar

The central Registrar holds the master list of all active conferences and their users. It responds to requests from Registrar Clients to add, delete and get lists of conferences and users.

*The Registrar implements no particular registration policy itself. It will happily obey any request from any Registrar Clients that connects to it, including requests to delete all conferences and users from its own master lists.*

### The Role of the Registrar Client

Registrar Clients are run by each user, and connect to the central Registrar. They are responsible for implementing a registration policy, and providing an interface to that policy.

#### Implementing a Registration Policy

Though a complete discussion on developing a registration policy is beyond the scope of this manual, a few comments are appropriate. Registration policies

should suit the needs of the group that will be using it, their membership, their goals, and their environment. Among some of the issues to be decided when developing a registration policy are the following:

- who can create conferences?
- will all conferences be visible to all users?
- who is allowed to join a conference?
- who can delete conferences?
- can users remove other users from conferences?
- can users force other users into a conference?

### **Building an Interface to the Policy**

Once the details have been fleshed out for a registration policy, then the design a user interface which implements that policy will be needed. Again, the key here is to keep in mind the needs of the group that will be using the interface (or several interfaces, depending on the policy).

## **Implementation Concepts**

### **Operations on the Registrar's Lists**

Any Registrar Client can operate on the Registrar's lists. There are six possible operations: adding a conference, deleting a conference, adding a user to a conference, deleting a user from a conference, retrieving a list of all conferences, and retrieving the list of users for a particular conference. GroupKit provides routines to help send these requests to the Registrar — they are described below under “Sending Messages to the Registrar.”

*Note that you will not receive notification of changes you make until you ask to retrieve the list of conferences or users.* Furthermore, you usually must wait for this notification rather than taking action immediately after sending the request — most operations in GroupKit require the unique conference or user number which is assigned by the Registrar in fulfilling “add” requests.

### **Reacting to Changes in the Registrar's Lists**

When a notification of changes (via events that are described in the “Registration Events” section below) is received, then the Registrar Client must decide if it wishes to honor these changes. Normally, any changes received by the Registrar Client *will* be honored, but for a particular policy it may be necessary to filter certain changes. For example, you may decide to filter out a change requesting the deletion of a conference if there are still users present in the conference.

The Registrar Client provides a “mirror” of the list contained in the Registrar. Yet changes from the Registrar are not automatically reflected in the Registrar Client's mirror. You must explicitly (via GroupKit-supplied routines, see “Supplied Event Handlers”) ask that these changes be incorporated into the local mirror. While normally you will ask that these changes be incorporated, filtering changes is as simple as not calling the routines that incorporate the changes.

When the Registrar Client incorporates a change from the Registrar, whether adding or deleting of a conference or a user, that change is said to be *approved*.

## Reacting to Approved Changes

Until changes are approved by the Registrar Client, the change should not be considered to have happened. So for example, a Registrar Client displaying a list of users should not update that display to include a new user until after the new user has been approved to join the conference.

GroupKit generates a set of events that reflect the various approvals: new conference, deleted conference, new user or deleted user. These are described in “Registration Events”, and are intended to allow you to update your interface when changes actually take effect.

## Spawning Conference Processes

GroupKit Registrar Clients are also responsible for creating new conference processes, and communicating relevant registration information to them (for example, that the process should join to another conference process on a particular machine).

A number of utilities (discussed in the “Miscellaneous Commands” section, below) are provided to help with keeping track of spawned processes.

---

# Sending Messages to the Registrar

GroupKit’s registration system provides a number of routines to send messages from a Registrar Client to the central Registrar.

## Requesting a new conference

To request that a new conference be created, use the *gk\_callNewConference* command. It takes as a parameter a keyed list specifying at least the following information about the conference:

---

confname	name of the conference
conftype	type of the conference
originator	host/port of the conference originator

---

The central Registrar will assign the unique conference number normally used to identify the conference. Note that changes will not be reflected until you ask for the list of conference (see “Requesting the list of known conferences” below).

This routine will not usually be called directly; the *gk\_createRequestedConf* handler (see “Supplied Event Handlers”) can call this as a result of the user asking to create a conference via GroupKit supplied registration interface. If the “Conferences” menu is not used then this routine will have to be called directly. Below is an example call. (Note that “gk\_uniqprogid” is discussed under “Miscellaneous Commands” later in this section).

```
keylset conf confname Joe conftype "Brainstorming Tool" \  
        originator [uniqprogid]  
gk_callNewConference $conf
```

### Requesting to delete a conference

To request a conference be deleted, use the *gk\_callDeleteConference* routine. It takes as a parameter the unique id number of the conference:

```
gk_callDeleteConference 3
```

### Requesting the list of known conferences

To request a list of known conferences, use the *gk\_pollConferences* routine. It takes no parameters. Any differences between the Registrar's list and the Registrar Client's copy will cause events to be generated (see "Registration Events" below).

```
gk_pollConferences
```

### Requesting a user be added to a conference

To request a new user be added to a conference, use the *gk\_callJoinConfWithKeys* command. It takes as parameter both the unique id number of the conference to be joined and a keyed list specifying at least the following information about the user:

---

userid	the login userid
host	the host of the user's Registrar Client
port	the port of the user's Registrar Client
username	the full name of the user

---

Note that the central Registrar will assign the unique user number normally used to identify the user. Note that changes will not be reflected until you ask for the list of users (see "Requesting the list of users in a conference" below).

```
keylset user userid [exec whoami] host [userprefs host] \  
port [userprefs port] username [userprefs name]  
gk_callJoinConfWithKeys 3 $user
```

A special form of this command is available when it is the local user (i.e. the one running the Registrar Client) that wants to join the conference. In that case use:

```
gk_callJoinConference 3
```

### Requesting a user be removed from a conference

To request a user be deleted from a conference, use the *gk\_callLeaveConference* routine. It takes as a parameter the unique id number of the conference and the user:

```
gk_callLeaveConference 3 5
```

### Requesting the list of users in a conference

To request a list of users in a particular conference, use the *gk\_pollUsers* routine. It takes as a parameter the unique id of the conference. Any differences between the Registrar's list and the Registrar Client's copy will cause events to be generated (see "Registration Events" below).

---

## Registration Environments

The Registrar Clients rely on two main environments for holding their information. The *confs* environment holds information about the various conferences and their users that are known to the central Registrar. The *spawned* environment by contrast, keeps information about conference processes that have been created. Typically Registrar Clients do not need to change this information directly, but instead can respond to events generated by GroupKit.

### The “confs” environment

The *confs* environment holds all the information about all conferences and their users; it is effectively a mirror of the environment maintained by the central registrar. Each conference is stored underneath the “c” key in the environment, according to its unique conference number, specified by the registrar, for example “confs c.1.confname.”

For each conference at least the following information is kept:

---

<i>prefix.confname</i>	name of the conference
<i>prefix.conftype</i>	type of the conference
<i>prefix.originator</i>	host/port of the conference originator

---

As well, a list of users is kept within the user’s field of each conference, again differentiated by unique user number, for example “confs c.1.users.2.username”. At least the following information is kept on each user:

---

<i>prefix.username</i>	name of the user
<i>prefix.userid</i>	login userid of the user
<i>prefix.host</i>	host of the user’s Registrar Client
<i>prefix.port</i>	port of the user’s Registrar Client

---

### The “spawned” environment

When Registrar Clients create a conference process (i.e. join the local user to the conference and start up the application on their screen), they use the spawned environment to keep track of several pieces of information about the conference process. This is used for example to send messages to the conference process.

The spawned environment, like *confs*, is again indexed by the unique conference number under the “c” key, so for example “spawned c.1.localuser” specifies the user number of the local user in the conference. The following information is kept on each spawned conference process:

---

<i>prefix.stillCheck</i>	“yes” if the process has not connected back yet
<i>prefix.fd</i>	socket of the connected conference process
<i>prefix.localuser</i>	user number of the local user in the conference
<i>prefix.pending</i>	list of commands to be sent to the conference

---

## Registration Events

This section describes the events that are generated by GroupKit’s registration system. When working through this section, keep in mind the discussion of “approval” of changes in the Overview. GroupKit’s supplied event handlers are described immediately after this section and provide some standard responses to many of these events.

### Changes to Conferences

Four events can be generated that relate to changes in the conference list: adding and deleting of conferences, as well as the corresponding “approval” events.

#### The “foundNewConf” event

This event is generated for each conference that is found to be in the Registrar’s lists, but not our own. The event specifies information about the conference. At least the following fields are guaranteed to be present. (See the *gk\_addConfToList* handler, discussed under “Approving New Conferences” in the next section).

---

type	foundNewConf
originator	host and port of conference’s originator
confnum	unique id of the conference
confname	name of the conference
conftype	type of the conference

---

#### The “newConfApproved” event

This event is generated when a new conference has been approved, at which time all its information is contained in the “confs” environment. A typical response might be to add the conference to a list of available conferences displayed to the user. Also, if the conference is one we created, many registration policies specify that the local user should be automatically joined to the conference.

---

type	newConfApproved
confnum	unique id of the approved conference

---

### The “foundDeletedConf” event

This event is generated for each conference that is found to be in the Registrar Client’s lists, but not the Registrar’s list (i.e. it has been removed). The event specifies the conference number. (See the *gk\_removeConfFromList* handler, discussed under “Approving Deleted Conferences” in the next section).

---

type	foundDeletedConf
confnum	unique id of the deleted conference

---

### The “deleteConfApproved” event

This event is generated when a conference deletion has been approved, at which time all its information has already been removed from the “confs” environment. A typical response might be to remove the conference from a list of available conferences displayed to the user.

---

type	deleteConfApproved
confnum	unique id of the deleted conference

---

### The “confListProcessed” event

This event is generated when the client has finished checking for new and deleted conferences and is, therefore, “consistent” with the registrar. This event is generated after every complete pass through the conference list (e.g.. from a cell to *gk\_pollconference*).

type	confListProcessed
------	-------------------

## Changes to Users

Similar to conferences, we have four main events can be generated that relate to changes in the user list for a conference: adding and deleting of users, as well as the corresponding “approval” events. There is one additional event, specifying that the last user has left the conference.

### The “foundNewUser” event

This event is generated for each user that is found to be in the Registrar’s lists, but not our own. The event specifies information about the user. At least the following fields are guaranteed to be present. (See the *gk\_addUserToList* handler, discussed under “Approving New Users” in the next section).

---

type	foundNewUser
confnum	the conference the new user wants to join
usernum	the unique user number for the user
userid	the login userid

host	the host running the user's Registrar Client
port	the port of the user's Registrar Client
username	the full name of the user

---

#### The "newUserApproved" event

This event is generated when a new user has been approved, at which time all its information is contained in the "confs" environment. A typical response might be to add the user to a list displayed to the user. Also, if the user is ourself, many registration policies specify that we should create the conference process.

type	newUserApproved
confnum	the conference the user is being added to
usernum	the unique user number for the new user

---

#### The "foundDeletedUser" event

This event is generated for each user that is found to be in the Registrar Client's lists, but not the Registrar's list (i.e. they have left the conference). The event specifies the conference and user numbers. (See the *gk\_removeUserFromList* handler, discussed under "Approving Deleted Users" in the next section).

type	foundDeletedUser
confnum	the conference the user is leaving
usernum	the user who is leaving

---

#### The "deleteUserApproved" event

This event is generated when a user deletion has been approved, at which time all its information has already been removed from the "confs" environment. A typical response might be to remove the user from a list displayed in the interface.

type	deleteUserApproved
confnum	the conference the user is leaving
usernum	the user who is leaving

---

#### The "lastUserLeftConf" event

This event is generated when the last user has left the conference. Some registration policies may dictate that this will cause the conference itself to be deleted.

type	lastUserLeftConf
confnum	the unique id of the conference

---

#### The "userListProcessed" event

This event is generated when the client has finished checking for new and deleted users and is therefore “consistent” with the registrar. This event is generated after every complete pass through the users list (e.g., from `gk_pollUsers` cells).

type	userListProcessed
confnum	the unique id of the conference

## Events about Conference Processes

### The “conferenceDied” event

When a socket connection belonging to one of the conference processes we spawned is closed (typically by the user quitting the conference), a *conferenceDied* event is generated. Depending on the registration policy, we may join the user back up to the conference, or more typically tell the central Registrar that they have left. (See the *gk\_userLeft* handler, discussed under “Handling Leaving Users” in the next section).

---

type	conferenceDied
conf	the unique id of the conference process that quit
user	the unique id of the local user in the conference
filedesc	the socket the process was connected on

---

### The “userRequestedNewConf” event

GroupKit provides utilities to allow the user to specify a conference to create through the user interface (see the “Miscellaneous Commands” section below). When the user does create a conference, a *userRequestedNewConf* event is generated. Typically, the response to this event is to create the requested conference. (See the *gk\_createRequestedConf* handler, discussed under “Creating Requested Conferences” in the next section).

---

type	userRequestedNewConf
confname	the name the user chose for the conference
conftype	the type of conference to be created
originator	host and port of the local Registrar Client

---

---

## Supplied Event Handlers

This section discusses the event handlers that GroupKit provides to respond to the various registration events. They provide the “normal” behavior for handling these events. They are typically called as the action part of a trigger for an event.

For simplicity in calling them, none of these event handlers take any parameters. Instead, they communicate via environments, created by the routine which generates the associated event. For example, there is a routine in rc.tcl called *gk\_foundNewConference* which generates the *foundNewConf* event, for adding a new conference to the Registrar Client's list. Before generating the event, an environment called *newconf* is created which holds information about the new conference. The normal handler for that event, *gk\_addConfToList*, expects that environment to be present so it can retrieve the needed information.

The upshot of all that is that if these routines are called from within the context of the appropriate events (via a trigger) things will work fine, but if you plan on calling them outside a trigger for whatever reason, you'll have to do some work to set up the appropriate environments before calling each routine. You'll have to RTFC (read the fabulous code) to discover these.

### **Approving New Conferences**

The standard action to take when a new conference is discovered (announced via a "foundNewConf" event) is to approve the conference. The *gk\_addConfToList* handler will do this, by adding the new conference to the Registrar Client's list, as well as sending out a *newConfApproved* event so that any changes may be reflected in the interface or other parts of the Registrar Client.

### **Approving Deleted Conferences**

The standard action to take when a deleted conference is discovered (announced via a "foundDeletedConf" event) is to approve the deletion. The *gk\_removeConfFromList* handler will do this, by removing the conference from the Registrar Client's list, as well as sending out a *deleteConfApproved* event so that any changes may be reflected in the interface or other parts of the Registrar Client.

### **Approving New Users**

The standard action to take when a new user is discovered (announced via a "foundNewUser" event) is to approve the user into the conference. The *gk\_addUserToList* handler will do this, by adding the new user to the Registrar Client's list, as well as sending out a *newUserApproved* event so that any changes may be reflected in the interface or other parts of the Registrar Client.

### **Approving Deleted Users**

The standard action to take when a deleted use is discovered (announced via a "foundDeletedUser" event) is to approve the deletion. The *gk\_removeUserFromList* handler will do this, by removing the user from the Registrar Client's list, as well as sending out a *deleteUserApproved* event so that any changes may be reflected in the interface or other parts of the Registrar Client.

### **Handling Leaving Users**

The standard action when a user leaves the conference (announced via a "conferenceDied" event) is to inform the central Registrar that the user has left. The *gk\_userLeft* handler sends the appropriate message.

*Note: The `gk_userLeft` handler currently has three parameters passed to it (`confnum`, `usernum`, `file descriptor`) rather than communicating via an environment. This will change.*

### Creating Requested Conferences

The standard action when a user requests to create a conference (announced via a “`userRequestedNewConf`” event) is to create the conference, which is done via `gk_createRequestedConf`.

---

## Summary of Environments, Events and Handlers

The following table summarizes the various registration events, their associated attributes, and the handlers that are supplied to provide default behaviors for them.

Event	Attributes	Handler
foundNewConf	originator confnum confname conftype	<code>gk_addConfToList</code>
newConfApproved	confnum	
foundDeletedConf	confnum	<code>gk_removeConfFromList</code>
deleteConfApproved	confnum	
foundNewUser	confnum usernum userid host port usernum	<code>gk_addUserToList</code>
newUserApproved	confnum usernum	
foundDeletedUser	confnum usernum	<code>gk_removeUserFromList</code>
deleteUserApproved	confnum usernum	
lastUserLeftConf	confnum	

conferenceDied	conf user filedesc	gk_userLeft
userRequestedNewConf	confname conftype originator	gk_createRequestedConf

---

## Miscellaneous Commands

This section describes the other facilities GroupKit provides to help building Registrar Clients.

### A User Interface to Create Conferences

GroupKit provides a menu containing all of the available conferences (read from the `.groupkitrc` file). You can add this menu to your Registrar Client to allow users to create conferences. When a user selects one of the conference types from the menu, a dialog box appears asking them to choose a name for the conference. An event (`userRequestedNewConf`) is then sent to your application.

To create this menu, use the `buildConferenceMenu` command when your menu is posted, for example:

```
menu .menubar.confmenu.conferences -postcommand \  
    "buildConferenceMenu .menubar.confmenu.conferences"
```

### Creating Conference Processes

GroupKit provides the `gk_createConference` command to create a conference process. You provide the conference number and the user number for the local user — other information is obtained from within the “confs” environment:

```
gk_createConference $confnum $usernum
```

You can send messages to a conference process via the `gk_toConf` command. Because there is a time delay between when a conference is created and when it connects back so that the Registrar Client can actually send messages to it, messages are queued in the “spawned” environment if necessary. There pending messages are automatically transmitted when the conference connects back to the Registrar Client.

```
gk_toConf $confnum $command
```

There are specific routines that tell our conference process to connect to one or more conference processes of other users (i.e. all other existing users).

```
gk_joinTo $usernum $confnum  
gk_joinToOthers $confnum $myusernum
```

## Keeping Track of Joined Conferences

The “spawned” environment is used to keep track of conferences we have already spawned (via `gk_createConference`, above). However, there is some time delay between when the user requests a conference to be created and when we actually create the conference process (at least one or two round trips to the Registrar). To prevent users from attempting to join a conference they’ve already asked to join to, GroupKit provides some utility functions your interface may use to keep track of which conferences its asked to join.

To specify that you’ve asked to join a particular conference, use:

```
gk_conferenceJoined $confnum
```

To say that you’re no longer joined to a particular conference, use:

```
gk_noLongerJoined $confnum
```

To get a list of the conferences you’re joined to, use:

```
gk_confsJoined
```

Finally, to find out if you’re joined to a particular conference, use:

```
gk_alreadyJoined $confnum
```

## Creating an Originator ID

GroupKit keeps track of which Registrar Client created which conferences via an “originator”. This originator is composed of the host and port number of the Registrar Client, and serves as a unique identifier for the Registrar Client. To generate this identifier, use the following:

```
gk_uniqprogid
```

---

## Putting it All Together

This section provides a rather detailed walk-through of the Open Registration client supplied with GroupKit.

### Initializing the Registrar Client

The first step is initializing the Registrar Client, which connects up to the Registrar, sets up a socket for conferences that we spawn to connect to us, and does some other housekeeping. We also ask the Registrar for a list of conferences to find out what is already around.

```
gk_initGenericRC $argv  
gk_pollConferences
```

### Specifying Registration Policy

We next begin to specify all the triggers that will implement the registration policy.

### **Add all new conferences**

Any new conferences we find are valid, so add them to the conference list.

```
gk_on { [gk_event type]=="foundNewConf" }
{gk_addConfToList}
```

### **Join conferences we create**

For any conferences that we create, we will join the local user to them.

*Note: This should be using the newConfApproved event rather than the foundNewConf event, and will be changed.*

```
gk_on { ([gk_event type]=="foundNewConf") && \
        ([gk_event originator]=="[uniqueprogid]") } \
{ gk_callJoinConference [gk_event confnum]; \
  gk_pollUsers [gk_event confnum] }
```

### **Add all new users**

Any new users we find are valid, so add them to the users list.

```
gk_on { [gk_event type]=="foundNewUser" }
{gk_addUserToList}
```

### **Create a conference process when we are added**

When we receive a foundNewUser event for ourselves, create the conference process and join to existing users of the conference.

```
gk_on { ([gk_event type]=="foundNewUser") && \
        ([gk_event host]=="[userprefs host]") && \
        ([gk_event port]=="[userprefs port]") } { \
  gk_createConference [gk_event confnum] \
    [gk_event usernum]; \
  gk_joinToOthers [gk_event confnum] \
    [gk_event usernum] \
}
```

### **Remove all deleted users**

All deleted users we find are valid, and are removed from the users list.

```
gk_on { [gk_event type]=="foundDeletedUser" } \
{gk_removeUserFromList}
```

### **Delete the conference when the last user leaves**

When the last user of a conference leaves, we tell the Registrar to delete the entire conference.

```
gk_on { [gk_event type]=="lastUserLeftConf" } \
{gk_callDeleteConference [gk_event confnum]; \
gk_pollConferences}
```

### Delete users of killed conference processes

When a conference process connected to us dies, report that the user has left the conference.

```
gk_on {[gk_event type]=="conferenceDied"} \
      {gk_userLeft [gk_event conf] [gk_event user] \
       [gk_event filedesc]}
```

### Remove all deleted conferences

All deleted conferences we find are valid, and are removed from the conference list.

```
gk_on {[gk_event type]=="foundDeletedConf"} \
      {gk_removeConfFromList}
```

### No longer joined when we are removed

If our user is removed from the conference, then we are no longer joined to it.

*Note: Again, this should be changed to an "approved" event.*

```
gk_on {[gk_event type]=="foundDeletedUser"}&&\
      ([gk_event usernum]==[spawned c.[gk_event \
       confnum].localuser])} \
      {gk_noLongerJoined [gk_event confnum]}
```

### Create all requested conferences

When a user asks to create a new conference create it as requested.

```
gk_on {[gk_event type]=="userRequestedNewConf"} \
      {gk_createRequestedConf}
```

## Building the Interface

### Join chosen Conference

Join to an existing conference, this routine is called in response to a double-click in the conferences listbox. First which conference was selected is determined. Then if we are not already joined to the conference we are joined now.

```
proc gk_doJoin confIndex { global openrc
  set confnum [gk_confFromListbox $confIndex]
  if {$confnum!=""} {
    if {![gk_alreadyJoined $confnum]} {
      gk_conferenceJoined $confnum
      gk_callJoinConference $confnum
      gk_pollUsers $confnum
    }
  }
}
```

## Define the help message and its title

This information will be added to the help Menu on the default menu bar for the registrar client.

```
set openrc(help_title) "Open Registration"

set openrc(help_content) {
{normal} {This is an example of a }
{normalitalic} {registration }
{normal} {system that lets you \
create and join conference applications.

}
{normalbold} {Seeing conferences and participants.
}
{normal} {The Conference pane shows all conferences on\
the system, whether created by you or by others.

Selecting a conference from the Conferences pane will \
list the people in it in the Participants pane.

}
{normalbold} {Creating a conference application.
}
{normal} {Select a conference application from the
Conferences menu.

A dialog box then gives you the opportunity to rename \
the conference if you wish, to proceed, or to cancel the
operation.

A window containing that conference application \
will appear, and its name is added to the Conferences
pane.

}
{normalbold} {Joining an existing conference.
}
{normal} {Double-click a conference listed in the \
Conferences pane. You will automatically join that \
conference if you are not already in it. The conference \
application will appear on your screen.

}
{normalbold} {Changing your name.
}
{normal} {The bottom field lets you change your name if
you \
wish. However, it will not affect conferences that you \
have already joined.

}
{normalbold} {Reloading the .groupkitrc
}
{normal} {Selecting Re-initialize from the File menu
will \
re-load the .groupkitrc file. This is usually done only
\
if you have changed the contents of your .groupkitrc on
\
the fly.
}}
}}
```

## Specify the main window

It contains two scrollable lists, one containing a list of all the conferences (their names actually) and the second a list of users for a particular conference (selected in the first list). Double clicking on a conference allows a user to join the conference. A Conferences menu lets the user create new conferences and assign them a name. The user's name is also displayed in the entry box at the bottom of the window.

```
wm title . $openrc(help_title)
set italic -Adobe-helvetica-bold-o-normal--*-140-*
```

### Scrollable List 1 - ConferenceList

```
frame .f2
pack append [frame .f2.conflist -relief flat] \
[label .f2.conflist.label -text Conferences \
-font $italic] top \
[scrollbar .f2.conflist.scroll -relief ridge \
-command ".f2.conflist.list yview"] {right fillly} \
[listbox .f2.conflist.list \
-yscroll ".f2.conflist.scroll set" \
-relief ridge -geometry 20x6 \
-exportselction false -setgrid 1] \
{left expand fill}
```

### Scrollable List 2 - Conference Users

```
pack append [frame .f2.userlist -relief flat] \
[label .f2.userlist.label -text Participants \
-font $italic] top \
[scrollbar .f2.userlist.scroll -relief flat \
-command ".f2.userlist.list yview"] {right fillly} \
[listbox .f2.userlist.list \
-yscroll ".f2.userlist.scroll set" \
-relief ridge -geometry 15x6 \
-exportselction false -setgrid 1] \
{left expand fill}
```

### Entry Widget - User's Name

```
set openrc(user_name) [userprefs name]
pack append [frame .username -relief flat] \
[label .username.label -text "You are: " \
-font $italic] left \
[entry .username.entry -relief sunken \
-textvariable openrc(user_name) ] \
{left fillx expand}
```

A routine for updating the user's Name entered in the entry widget.

```
proc gk_changeUserName args {
    global openrc
    userprefs name $openrc(user_name)
}
```

## Constructing and altering to the default Menu bar

An easy method of building a menubar is to use the GroupKit default menu bar and make the necessary changes. In this case since we do not need the

collaboration menu it is removed. However, a conference menu is required so it is added to the menu bar and additional help information is added to the help menu.

```
gk_defaultMenu .menubar false
```

Adding the help information to the help menu

```
.menubar itemcommand help add command \  
-label "$openrc(help_title)" \  
-command "gk_topicWindow .helpWindow \  
-title {$openrc(help_title)} \  
-text {$openrc(help_content)}"
```

Adding an item to the file menu that re-reads the .groupkitrc file which in effect changes the environment to reflect the current contents of the .groupkitrc file.

```
.menubar itemcommand file add separator  
.menubar itemcommand file add command \  
-label "Re-initialize" \  
-command "userprefs delete prog; source  
~/groupkitrc"  
.menubar itemcommand file entryconfigure 0 \  
-command openrc_quit
```

A menubutton that will contain all the available conference and constructs a pulldown conference menu on the fly. Thus, it contains the current list of conferences.

```
.menubar add confmenu 1 -text Conferences  
.menubar itemconfigure confmenu \  
-postcommand "buildConferenceMenu  
.menubar.confmenu.menu"
```

Remove the Collaboration menu from the default menubar.

```
.menubar delete collaboration
```

## Build the Screen

Here the various widgets are positioned in the appropriate spots in the window.

```
pack .menubar -side top -fill x  
pack .f2 -side top  
pack .f2.conflist -side left  
pack .f2.userlist -side right  
pack .username -side left -fill x -expand yes  
  
bind .f2.conflist.list <1> \  
{%W select from [%W nearest %y]; _gk_updateUserList  
blat}  
bind .f2.conflist.list <Double-1> \  
{gk_doJoin [.f2.conflist.list curselection]}  
tk_listboxSingleSelect .f2.conflist.list
```

## Monitoring Conferences.

We need to watch when new or leaving users and conferences are approved so that the conference lists can be updated appropriately. First if a new conference has been started then it is added to the conference listbox. and the next trigger removes a conference from the listbox if all of the users have left. Similarly, the last two triggers add and remove users from the user listbox.

```
gk_on { [gk_event type]=="newConfApproved" } \
  { _gk_addConfToListbox [gk_event confnum] }
gk_on { [gk_event type]=="deleteConfApproved" } \
  { _gk_delConfFromListbox [gk_event confnum] }
gk_on { [gk_event type]=="newUserApproved" } \
  { _gk_addUserToListbox [gk_event confnum] \
    [gk_event usernum] }
gk_on { [gk_event type]=="deleteUserApproved" } \
  { _gk_delUserFromListbox [gk_event confnum] \
    [gk_event usernum] }
```

## Routines for maintaining the lists

Initialize the list of conferences to an empty string.

```
set openrc(listboxconfs) ""
```

Find the conference that was selected from the listbox. The first routine returns the conference at a given position in the listbox, the second one returns the currently selected conference.

```
proc gk_confFromListbox index { global openrc
  if { [catch {set urgh [lindex $openrc(listboxconfs) \
    $index]}] ==0 } {
    return $urgh
  }
  return ""
}

proc gk_selectedConference {} {
  return [gk_confFromListbox [.f2.conflist.list \
    curselection]]
}
```

Add a conference to the list The “confnum” is added to the conference list and the conference name is appended to the conference listbox.

```
proc _gk_addConfToListbox confnum { global openrc
  lappend openrc(listboxconfs) $confnum
  .f2.conflist.list insert end [confs
  c.$confnum.confname]
  _gk_updateUserList
}
```

Delete a conference from the list. Given a particular “confnum” the conference is removed from the listbox and the conference list is updated to reflect this change.

```
proc _gk_delConfFromListbox confnum { global openrc
  set idx [lsearch $openrc(listboxconfs) $confnum]
```

```

        set openrc(listboxconfs) [lreplace
$openrc(listboxconfs) $idx $idx]
        .f2.conflist.list delete $idx
        _gk_updateUserList
    }

```

Add a User. If the given new user is a participant in the selected conference then place the user in the listbox.

```

proc _gk_addUserToListbox {confnum usernum} {
    if {$confnum==[gk_selectedConference]} {
        _gk_updateUserList
    }
}

```

Delete a user. If the user leaving the conference is in the selected conference then remove them from the users listbox.

```

proc _gk_delUserFromListbox {confnum usernum} {
    if {$confnum==[gk_selectedConference]} {
        after 1 _gk_updateUserList
    }
}

```

Update the display of users for the currently selected conference. If the selected conference changes then the users listbox is updated to reflect the users in the newly chosen listbox. The environment “confs” is used to retrieve the list of the users in the conference

```

proc _gk_updateUserList args {
    global openrc

    set confnum [gk_selectedConference]
    .f2.userlist.list delete 0 end
    if {$confnum!=""} {
        foreach i [confs keys c.$confnum.users] {
            .f2.userlist.list insert end \
            [confs c.$confnum.users.$i.username]
        }
    }
}

```

# Index

addEnvInfo 24, 56, 57

changeEnvInfo 24, 27, 56, 57

Collaboration 11, 39, 40, 76

conference 14, 67

conference application 9, 12, 18, 25, 40, 41, 51, 59

Conference Applications 8

event 11, 16, 18

events 16, 17, 18, 57

gkViewport. 47

gk\_attributes 41

gk\_callDeleteConference 62, 72

gk\_callNewConference 61

gk\_createRequestedConf 61

gk\_defaultMenu 12, 18, 25, 39, 51, 55, 76

gk\_gestaltView 48

gk\_groupScroll 44, 45, 53

gk\_initConf 12, 18, 25, 45, 46, 47, 51, 55

gk\_initializeTelepointers 46, 47, 56

gk\_newenv 20, 22, 24, 25, 26, 27, 55

gk\_on 16, 17, 19, 24, 27, 54, 57, 72, 73, 77

gk\_participants 40

gk\_pollConferences 62, 71, 72

gk\_postEvent 17, 18, 19

gk\_specializeWidgetTreeTelepointer 46, 47, 56

gk\_toAll 13, 14, 15, 18, 19, 53, 54, 55

gk\_toOthers 15

gk\_topicWindow 13, 43, 44, 52, 76

gk\_toUserNum 15, 54

gk\_viewport 49, 53

groupkitrc 22

Help 11, 12, 39, 42, 43, 52, 74, 76

keyed list 17, 19, 21, 61, 62

local user 13, 14, 16, 19, 23, 48, 51, 53, 56, 62, 63, 64, 70, 72

newUserArrived 17, 19

Registrar 7, 8, 11, 23, 26, 59, 60, 61, 62, 63, 64, 65, 66, 67, 71

Registrar Client 7, 8, 10, 51, 59, 60, 61, 62, 63, 65, 66, 68, 70, 71, 74

Registrar Clients 59

remote users 15, 19, 21, 23

Triggers 16, 17, 18, 56, 57, 71, 77

updateEntrant 16, 18, 19, 54

userDeleted 18, 19

userprefs 11