GroupKit 5.0 Documentation

Overview

- GroupKit 5.0 Overview
- Internal Architecture

User Manual

- Introduction
- Using GroupKit Applications - An End User's View
- An Example GroupKit Program
- Coordinating Multiple Users
- Using Environments to Find out Information about Users
- Conference Events
- Groupware Widgets and Awareness
- Shared Environments
- Concurrency Control
- Debugging and Performance Issues
- Other GroupKit Features

(Primarily) External Commands Reference

- gk::to - Multicast remote procedure calls
- gk::event - Generate and respond to local events
- gk::schedule - Manage the execution of repetitive commands
- gk::environment - Shared tree-based data structure
- gk::users - Keep track of information on conference users
- gk::conference - Miscellaneous conference facilities
- gk::sessionmgr - Utilities to build session manager applications
- gk::preferences - Manage user preference information

User Interface and Widget Commands Reference

- gk::telepointers - Multiple cursors
- gk::scrollbar - Multi-user scrollbar

- gk::radar - Radar view awareness display
- gk::menus - Standard menubar facilities for GroupKit applications
- gk::participants - Various facilities for interacting with user information

(Primarily) Internal Commands Reference

- gk::rpc - Maintain socket connections between processes
- gk::debug - Debugging facilities
- gk::registry - Convenience routines for accessing the global registry
- gk::init - Handle process startup and initialization
- gk::shutdown - Handle process termination and cleanup
- gk::service - Locate and connect to specific facilities offered by other processes
- gk::awareness - Support for maintaining workspace awareness information
- gk::base64 - Base64 encoding and decoding of binary files

Supplied Services

- idgenerator - Assign unique identifiers
- environmenthost - Host centralized environments
- laucher - Create new conference processes
- persistence - Provide a persistence repository for conference data

---

Last updated June 28, 1998 by Mark Roseman.

GroupKit 5.0 Overview

Introduction

GroupKit 5.0 is a substantial rewrite, loosely based on the design found in the unreleased GroupKit 4.0. The GroupKit API has been completely changed, and now relies on the namespace mechanism introduced in Tcl 8.0. Internally, the code has also been cleaned up to conform with Tcl coding conventions. GroupKit 5.0 supports Unix, Windows and Macintosh.

GroupKit 4.0 was an experimental version, intended to test the applicability of a meta-architecture for GroupKit. We discontinued development of GroupKit 4.0, but retained many of the ideas and concepts in GroupKit 5.0. Additionally, we had the goals of making this release very robust, stable and maintainable.

Meta-Architecture

GroupKit 5.0 keeps many of the core concepts from the meta-architecture developed by Ted O'Grady for GroupKit 4.0. In particular, the internals of GroupKit are still highly environment-based, and most communications is routed through environments. In general, the meta-architecture has been somewhat simplified, and some of the generality removed.

Services and registries are still supported, and interactions between environments is still configurable. The mechanism to do so has been changed, generalizing the previous mechanism which is now based on the metaphor of routers and subnets.

Details on the meta-architecture will be provided at a later date.

New API

The API in GroupKit 5.0 is completely different from that in previous versions, which necessarily means your code will break. There were several guiding principles in designing the new API:

- the new namespace mechanism introduced in Tcl 8.0 was used, such that everything to do with GroupKit is now stored within the "gk" namespace
- similar commands were grouped together into a single command, where the first parameter of the command identifies the particular command to be invoked; this is similar to the "string" or "file" commands in Tcl
- the overall number of commands was reduced, and it was clearly emphasized within the code which were internal and which were external commands

The following table provides a mapping from commands used in previous versions of GroupKit to commands in the new version:

	Old API	New API
RPCs	gk_toAll *cmd ?args?*	gk::to all *cmd ?args?*
	gk_toOthers *cmd ?args?*	gk::to others *cmd ?args?*
	gk_toUserNum *usernum cmd ?args?*	gk::to *usernum cmd ?args?*
	gk_serialize *cmd ?args?*	gk::to all -serialize *cmd ?args?*
Environments	gk_newenv *?options? envName*	gk::environment *?options? name*
Events	gk_bind *event action*	gk::event bind *event action*
	gk_delbind *binding*	gk::event delete *binding*
	gk_notify *event params*	gk::event notify *event params*
Widgets	gk_addStandardMenus *?menubar*	gk::menus addstandard *?menubar?*
	gk_addMenuHtmlHelp *title html*	gk::menus addhelp *title html*
	gk_scrollbar *window ?options?*	gk::scrollbar *window ?options?*
	gk_initializeTelepointers	--
	gk_specializeWidgetTreeTelepointer *window*	gk::telepointers attach *window*
Utilities	gk_schedule *tag command ?interval?*	gk::schedule *tag command ?interval?*
	gk_cancelScheduled *tag*	gk::schedule cancel *tag*
	gk_amOriginator	gk::conference originator
	users *cmd ?args?*	gk::users *cmd ?args?*
	gk_info *cmd ?args?*	gk::info *cmd ?args?*
	gk_initConf	--

Coding Style and Conventions

All code in the GroupKit library conforms to the coding conventions found in the Tcl/Tk Engineering Manual (C code) and the Tcl/Tk Style Guide (Tcl scripts). You should find that the GroupKit code is now both consistent and documented, making it easier to understand and modify. If you have code that you are planning to contribute to GroupKit, please ensure that it follows these conventions.

Last updated February 26, 1998 by Mark Roseman.

Internal Architecture

Introduction

Okay, you know you're in trouble if you have to be looking in here. GroupKit's internals are pretty twisted, and if there's any way to avoid having to figure them out, I would really strongly suggest taking advantage of that opportunity right now.

Still here I guess. The first step, before getting into the details of this all, is to try to explain why things are so incredibly complex and what we were trying to accomplish.

As usual, it all comes down to flexibility. We'd long been advocates of making groupware customizable and personalizable for end users. This naturally extended itself to making the toolkit flexible enough so that developers can provide for the flexibility their end users needed. We knew that very little could be set in stone, particularly issues like session management, and how data is shared between processes.

Ted O'Grady did his Masters thesis on applying a technique called *open implementations* to groupware toolkits. Essentially the underlying idea is that not only is it enough for toolkits to provide an API that developers can use to build their applications, but a second API has to be provided to let developers change how that first API behaves; essentially providing a structured way to let application developers rewrite parts of the toolkit when they find that some of the assumptions made by the toolkit's developers don't fly for their particular application.

In GroupKit, this has focused (after too many iterations to count) on letting developers radically redefine the way that environments behave. As you'll see, environments are the core mechanism for doing anything inside GroupKit. They do a lot of different things. The reason they're able to do so many different things is that they are incredibly customizable.

The rest of this architecture overview consists of two parts. The first looks at how GroupKit *uses* environments and other mechanisms internally to provide its run-time infrastructure. From this you should learn enough to understand how GroupKit actually works, and how to make some basic changes. The second part talks about how to customize environments; if you need to make some really radical changes, you'll probably need to work through this part. Needless to say, for all of this you should be intimately familiar with the user-level facilities provided by GroupKit.

GroupKit's Run-Time Infrastructure

This section will look at how the existing GroupKit infrastructure is constructed. We'll identify the different types of processes used in GroupKit, talk about the registry which is the central directory of everything, talk about how new processes are created, how things get cleaned up when processes are destroyed, look at session management, and finally consider how broadcasts work.

Processes

Processes in GroupKit are essentially applications. Most of the time they correspond with actual operating system processes, but not always. For example, on the Mac, a session manager and several conferences may all run in a single operating system process, each within its own Tcl interpreter.

There are essentially three types of processes you'll find running in GroupKit. The first is a central process called the registrar; there's usually only one of these. It's most important job is to maintain the global registry; it also provides a number of other important services such as generating unique id's and hosting central environments.

The second type of process is the session manager, of which there are usually one per active user of the system. The main job of the session managers is starting and joining conferences.

Finally, conference processes run the actual GroupKit tools that most people spend their time using.

Every process is identified with a unique id number. The registrar is normally "0". Other processes get their id number from the "idgen" service that runs on the registrar, which may look something like "pumori_cpsc_ucalgary_ca>>>9357+3".

In the current implementation, processes are interconnected by sockets as follows. Session managers maintain a single socket connection to the registrar (i.e. client-server). Therefore, if session managers wish to communicate with each other, they normally do so via the registrar. Conferences maintain a socket to the registrar, a socket to the session manager that created them, and direct socket connections to every other conference process in their session (e.g. if three people have joined the same brainstorming session, each conference process has sockets to the two others). So conferences communicate to each other in a peer-peer fashion, and also have have connections to the central registrar and their session manager.

The Registry

One of the most important jobs of the central registrar is to maintain the global registry. A registry is just an environment that happens to have its data structured a particular way. It is used to look up system information. For example, if a process wants to locate a service it can look up its location in the registry. Similarly, to find an environment to connect up to, a user's name, or all the processes shared by a user, check the registry.

Specifically, GroupKit's global registry is a server environment named ::gk::globalRegistry hosted by the registrar. One of the best ways to understand all the pieces of GroupKit is to examine this environment. To do so, start the registrar and a session manager. Open up the "Debug" window in the session manager, and invoke the command "::gk::globalRegistry debug".

The registry tree is divided into a number of sections. We'll explore each of these in turn.

Processes

Each process has an entry under the "processes" key. This is indexed by the process id, so for example you'd find information on the registrar process under. "::gk::globalRegistry processes.0". The information you'll find stored here is the "host" the process is running on, the "port" where the process is running a socket listener, the "usernum" (which is redundant; this is the same as the process id), and the "group", which is described below.

For example, here is the processes tree of the registry showing the registrar, one session manager, and a conference created by that session manager, all which happen to be running on the same machine:

```
::gk::globalRegistry
   processes
      0
         host: ratbert
         port: 9357
         usernum: 0
         group: 0
      ratbert>>>9357+2
         host: ratbert
         port: 1108
         usernum: ratbert>>>9357+2
         group: ratbert>>>9357+2
      ratbert>>>9357+4
         host: ratbert
         port: 1110
         usernum: ratbert>>>9357+4
         group: ratbert>>>9357+2
```

Groups

Every process in GroupKit created by the same user's session shares the same group, e.g. a session manager and all conferences started from that session manager. The purpose of a group is to hold information like a user's

name, color, phone number etc. in one place rather than having multiple copies associated with each process. Not only is this more efficient, but it also means if the information changes, all processes automatically pick up the information.

Each group has a unique id (which is what is stored in the "group" field in the processes tree). In the current implementation, the process id of the session manager is also used as the group id. Information on each group is stored under the "groups" key of the global registry, indexed by the group id.

For each group, GroupKit stores the person's name ("username"), a color associated with them ("color"), and various information that is normally associated with their business card ("title", "dept", "company", "phone", "fax", "email", "www", and "office"). Finally, under the "members" key, GroupKit stores a list of all the processes that are members of the group.

Here is an example showing the group holding the session manager and conference processes from the above example:

```
::gk::globalRegistry
   groups
      ratbert>>>9357+2
         username: Mark Roseman
         color: red
         title: Reluctant Documenter
         dept: Computer Science
         company: University of Calgary
         phone: 403-220-7259
         fax: 403-284-4707
         email: roseman@cpsc.ucalgary.ca
         www: http://www.teamwave.com/~roseman/
         office: Math Science 618
         members:
            ratbert>>>9357+2: ratbert>>>9357+2
            ratbert>>>9357+4: ratbert>>>9357+4
```

Environments

The registry also stores information about shared environments under the "environments" key. In particular, it keeps enough information so that processes can connect to an environment started by another process. This includes the type of environment, e.g. "server" for client-server based environments, and a list of one or more environment "maintainers", which are processes that maintain the environment; to connect to the environment a process would connect to one of the maintainers. While a centralized environment will have only one maintainer, a peer-peer environment may have several, because you can connect to any peer to start using the environment.

The registrar starts up a number of centralized environments which it maintains, as shown in this registry fragment:

```
::gk::globalRegistry
   environments
      ::gk::globalRegistry
         type: server
         maintainers
            0: 0
      ::gk::sessionManagers
         type: server
         maintainers
            0: 0
```

Note that peer-peer environments associated with a particular conference are usually *not* stored in the global

registry, but in another environment specifically for the conference. We'll come to this shortly, but it again illustrates the fact that any environment can be used as a registry.

Services

The global registry also stores information about service providers, allowing service subscribers to locate appropriate services to subscribe to. These are stored under the "services" key. From there, services are indexed by the type of service, and within each type by a unique id number generated internally by the service code. Each service stores information about its name ("name"), and the process that is providing the service ("processID").

In this fragment, we see two services provided by the registrar, and a "launcher" service offered by a session manager:

```
::gk::globalRegistry
   services
      idgenerator
         0x1
            name: global
            processID: 0
      environmenthost
         0x2
            name: global
            processID: 0
      launcher
         ratbert>>>9357+2x2
            name: launcherratbert>>>9357+2
            processID: ratbert>>>9357+2
```

Starting New Processes

We'll briefly mention here how a new process gets started and integrates itself into the GroupKit world. On first startup, the process connects to the central registrar, and uses its idgenerator service to generate a process id for itself. It then connects to the global registry, and adds information about itself and its group (using the convenience routines provided by the "gk::registry" command).

Things are essentially the same when a new process is created by an existing process, which is done via a "launcher" service. The launcher passes a number of parameters to the newly created process. For example, the group id is provided, so that the new process becomes part of an existing group, rather than creating its own. Also, the new process connects back to the process launcher service, passing its process id to signify that it was created successfully.

Notice the inconsistency two paragraphs up? The new process connects to the idgenerator service to look up its id, but to do that it would need to look up the service in the global registry. Yet before it can connect to the registry it needs its id. Oops. The way around this bootstrap problem is using the "-socket" option of the "service subscriber" command which just assumes that the requested service exists on the other end of a given socket, and doesn't bother to look it up in the registry.

Cleanup

When processes are shutdown, there is a lot of cleanup work to be done, in particular getting rid of information about the process stored in the registry and other environments throughout the system.

This is actually handled very cleanly using the "addDependent" command of environments. Essentially, this lets you mark a particular node (or subtree) of an environment as dependent on the existence of a process, and when the process goes away, the corresponding nodes will be deleted automatically.

You'll find information about dependent nodes stored in most shared environments under the "_dependents"

key. (The leading underscore is a bit of a hack in the environments code; essentially any nodes that start with it won't show up in a traversal using the "keys" command, but the data is still replicated around between instances of environments, which wouldn't be the case if stored under the "option" toplevel key).

Here is an example showing dependents, again with the registrar, one session manager, and one conference running. Items are indexed by process, then by "data" or "option", and within that just by a unique id.

```
::gk::globalRegistry
   _dependents
      0
         data
            2: environments.::gk::globalRegistry.maintainers.0
            3: environments.::gk::sessionManagers.maintainers.0
            4: environments.::gk::confs.maintainers 0
            5: services.idgenerator.0x1
            6: services.environmenthost.0x2
            11: environments.::gk::confs.ratbert>>>9357+3.maintainers.0
      ratbert>>>9357+2
         option
            8: subnet.defaultserver.clients.ratbert>>>9357+2
         data
            2: groups.ratbert>>>9357+2.members.ratbert>>>9357+2
            5: services.launcher.ratbert>>>9357+2x2
      ratbert>>>9357+4
         option
            13: subnet.defaultserver.clients.ratbert>>>9357+4
         data
            2: groups.ratbert>>>9357+2.members.ratbert>>>9357+4
```

Broadcasts

Broadcasts in GroupKit are also based on environments. The "gk::to" command is initialized (using "gk::to configure environment") to send all broadcasts through a particular environment.

Environments have a subcommand called "execute" which specifies that the parameters to the subcommand should be evaluated as a Tcl command. The "gk::to" command uses the environment's router (described in excruciating detail below), to run this "execute" subcommand on all of the networked instances of the environment.

This means that if the environment in question uses a replicated architecture, the "gk::to" messages will be distributed in a replicated fashion. If it uses a centralized architecture, broadcasts will be sent around in a centralized fashion. The distribution of broadcasts follows the topology set up by the environment, whatever that may be.

Session Management Support

The GroupKit infrastructure provides certain facilities for supporting session management. The primary interface to these is the "gk::sessionmgr" command, which provides routines for creating and joining conferences.

Certain environments, maintained by the registrar process, are also available. The "gk::sessionManagers" environment is used to manage broadcasts, as described in the previous section.

The "gk::confs" environment simply keeps a list of the currently known conferences. Changes to this environment cause the session manager code to generate higher-level events describing exact changes.

When a conference is created, an environment named "gk::confs.$confid" (where $confid is the conference number) is created on the registrar. This environment contains information pertinent to the conference. It holds

the current list of users of the conference for example. It also acts as a registry for any environments created to be used strictly within the conference. As an example, the "gk::awarenessModel" environment is registered with the "gk::confs.$confid" environment, rather than the "gk::globalRegistry".

Building New Environments

This section discusses how to build entirely new types of environments. You might build a new environment to support a new type of network topology, to change the behavior of messages (for example to add support for a concurrency control strategy), or to pretty much do anything else you wanted.

There are several ways that you can customize environments, depending on exactly what you want to accomplish. These vary from adding or changing individual environment commands, changing what processes those commands are sent to, assembling new network topologies from existing components, or even building entirely new types of network configurations.

Adding and Redefining Environment Commands

The simplest thing you can do is redefine what an existing command does, or create a new one. To do this, you use the "command" option of an environment. If you for example wanted to add a new primitive allowing you to lock the environment, you could do so like:

```
$env command set lock myLockCmdHandler

proc myLockCmdHandler {env cmd args} {
   ...
}
```

You can also redefine the behavior of the existing commands. This is similar to subclassing them. For example, you could redefine the "set" command, adding functionality but still calling the original, as follows:

```
$env command rename set _originalset
$env command set set myNewSetHandler

proc myNewSetHandler {env cmd key value} {
   ...
   $env _originalset $key $value
   ...
}
```

Command Routing

The second thing you can do is change what processes an environment command is executed in. Normally, for example, if you call "$env set x y" this command is executed on the copy of the environment found in the local process. This can be easily changed however to execute in any of the copies of a shared environment running in different processes.

Each environment has a "router" built into it. The router is responsible for delivering messages to different instances of the shared environment, located among different processes. In particular, a router knows how to send messages to "all" (all instances of the shared environment), "others" (instances of the shared environment except the instance in the local process), and to a particular process, identified by its unique process id. Note that the actual mechanism by which the messages get sent is a separate issue (see the next section); in calling the router you just specify *who* the messages should go to, not *how*.

To specify that a particular message should be routed, use the "command routing" option for environments. For example, this code specifies that when invoked locally, the set, delete and execute commands should be sent to all processes containing an instance of the shared environment:

```
foreach i "set delete execute" {
    $env command routing $i all
}
```

Defining Routes

While the "command routing" option deals with who commands should be sent to, this section discusses how the environment's router can be configured to specify how the messages get sent.

Routers are set up to include one or more attached subnets. Each subnet represents a particular type of network topology. Three pre-made subnets are provided in GroupKit, and others can be defined (see later section).

A "server" subnet resides on an environment in a centralized process, and accepts connections from clients. This subnet therefore has direct knowledge of all its attached clients, and knows how to route messages directly to any of them. A "client" subnet would reside in the client processes, and knows only about the server; to route messages to everyone, it just sends them to the server. Finally, a "peer" subnet is used to establish a fully-replicated topology with no centralized component, where all peers communicate directly with each other.

When setting up a router, each subnet is given a name. This is so you could for example have several subnets of the same type on one router. For example, the router could act as a bridge between two replicated networks by including two different peer subnets.

You can add new subnets using the "router addsubnet" command. This takes the name to assign to a subnet, as well as a handler which implements the subnet. The handler may also take other parameters; for example the handler for "peer" subnets can optionally take the host and port of an existing peer to connect to. Here are two examples:

```
$env router addsubnet defaultserver gk::EnvServerSubnet

$env router addsubnet defaultpeer gk::EnvPeerSubnet $host $port
```

Note again that the router itself knows nothing about the internals of the subnets; this is completely encapsulated in their handlers. By calling the handlers, the routers can find out enough information (e.g. is a particular process located on a given subnet?) to route messages to and between different subnets. However, by combining together individual subnet components, arbitrary network architectures can be created.

Defining New Subnets

The three provided subnet types (server, client, peer) should be sufficient for most applications, particularly when combined in different ways. However, if you do need to build a new type of subnet, it is possible to do so.

As mentioned above, to define a new subnet type you need to provide a handler. This is a standard Tcl procedure accepting several parameters. The first is the environment for which this subnet will be a part of, the second is the name given to the particular subnet (remember that several subnets may be added to the router of a single environment), and the third parameter is an operation for the handler to perform. Any additional parameters required by the operation will follow in further parameters.

The first operation to be implemented is "init", which is called when the subnet is first added. You can do anything you want to initialize the subnet here. Any parameters provided by the "router addsubnet" command are passed along here.

The "route" operation is invoked by the router to specify that the subnet should send a message. It takes the following parameters: who the message should be sent to, the message itself, and an optional parameter specifying who this message was received from. The destination of the message will be specified as either "others" (all other processes known by the subnet), "notsender" (all processes except the sender of the message, as specified in the third parameter), or a process id meaning send the message to only that single process. The "notsender" tag essentially helps prevent loops in the routing process.

The third operation is "contains" which is called by the router to determine if a particular process (identified by process id, the only parameter here) can be reached via this particular subnet. It is used by the router for

efficiency purposes. The valid responses by the subnet to this query are "yes", "no", or "maybe".

Finally, the "connectfrom" operation is invoked when a subnet on a remote environment attempts to make a connection to the local subnet. You will be passed two parameters, the id of the remote process, and a socket used to connect to the remote process.

Those are all the operations a subnet needs to support. In terms of actual implementation, subnets rely heavily on the "gk::rpc" command to build their communications infrastructure and send messages around. Subnets typically store any necessary housekeeping information in the "option" tree of their environment, under the key "subnet.$subnetname".

Environment Types

The previous sections have talked about individual pieces. This section brings together the pieces, which is necessary to create a new type of environment that can be instantiated. This can combine any or all of redefining environment commands, specifying command routing, and adding subnets.

To define an actual environment type, you need to define a procedure that implements the new type; this procedure takes one parameter, the name of the environment which is to have that type applied to it. You also need to register this procedure, so that the gk::environment command knows about the new type.

Here is a fairly simple example, which implements a server environment type. It contains a single server subnet (which clients can connect to), and all modify commands are routed to everyone. Information about the environment is also stored in the registry, which a client environment type might query to find out how to connect. Users could instantiate an instance of this type of environment using "gk::environment -server".

```
proc gk::EnvTypeServer {env} {
    $env router addsubnet defaultserver gk::EnvServerSubnet
    foreach i "set delete execute" {
        $env command routing $i all
    }
    registry addenvironment $env server
}

gk::EnvironmentRegisterType -server gk::EnvTypeServer
```

What Now?

The preceeding sections should give you a bit of an overview of the process of creating new environment types. Your best bet is to look through the existing implementations of subnets and types within the core of GroupKit to get a better understanding of what is involved.

GroupKit User Manual - Introduction

"Hmmm, that's interesting. Hey Linda, take a look at this."

"What do you have there Carl? Oh," Linda asked, glancing at the schematic on the screen.

"I want to get the signal here", Carl pointed, "over to this other part here, but I'm not sure the best way with all this stuff in between."

"Why don't you route it this way?" Linda suggested, drawing a rough path along the schematic, "this should be the most effective."

"Oh and then I can move this chip over to here instead," Carl replied as he dragged the chip to its new location.

Linda moved a couple of other components over. "That looks like it should pretty much do it."

"Great, that'll work. Thanks for your help."

They both turned back to their own work. A pretty typical situation that repeats itself thousands of times daily in workplaces everywhere.

Except that Linda and Carl are thousands of miles apart.

What is Groupware?

Linda and Carl are using a technology called groupware, which lets people who are far apart but connected by a network collaborate together on the same documents - such as the schematic in this example - at the same time. Although groupware had its beginnings with the visionary work of Douglas Engelbart in the 1960s, only recently have networks and workstations become powerful and ubiquitous enough to truly support it. Some examples of groupware systems in use today include shared electronic whiteboards, multi-user text editors, tools to support group brainstorming, and of course a wide variety of multi-player games.

Groupware actually refers to any technology that lets people work together. So things like email and Usenet news are also rudimentary groupware technologies. The difference is that email and Usenet don't allow people to work together at the same time or "synchronously", but instead support different time or "asynchronous" work. Another distinction between types of groupware is whether the system supports people working in the same place (such as a team meeting room) or at a distance, such as different sites on the Internet. We're mostly concerned with groupware for

geographically distributed groups working together synchronously.

Unfortunately, building groupware applications can be extremely difficult. Implementing even the simplest system is a lengthy and tedious process. Every application must worry about creating and managing socket connections, parsing and dispatching inter-process communication, locating other users on a network and connecting to them, keeping shared resources consistent between users, and so on. Using conventional programming tools, a lot of low level code must be written before getting to the specifics of the application. Groupware is an application domain crying out for better programming tools.

What is GroupKit?

GroupKit is an extension to Tcl/Tk that makes it easy to develop groupware applications to support real-time, distance-separated collaborative work between two or more people. Using Tcl's built-in socket commands for its low level networking, GroupKit provides an application framework that handles most details of building groupware automatically for you, so you can spend time just writing your application rather than its groupware infrastructure.

GroupKit grew out of our frustrations in building groupware systems without proper tools to support the job. By moving the common elements of groupware into a Tcl/Tk extension, we can now create programs in three days that originally took three months to write, and whose complexity shrunk from several thousand lines of code to only a few hundred lines. We've also found that it is often easy to take an existing single-user Tcl/Tk program and convert it to a multi-user program. With GroupKit, we're finding that writing groupware is only slightly harder than writing an equivalent single-user program.

The GroupKit distribution consists of the GroupKit extension itself (which is implemented as a mixture of Tcl and C), documentation, and approximately thirty example groupware applications.

GroupKit User Manual - An End User's View

Before delving into the details of how to build GroupKit applications, we'll first walk through how an end user would run some of the existing applications included in the distribution. This will give you a chance to make sure things are set up on your own system, as well as introduce some important concepts about groupware programs.

We'll assume that GroupKit has already been compiled and installed on all participating systems, with a registrar process running. If GroupKit has not yet been installed, you can install it yourself by following the step by step installation instructions in the README file in the software distribution.

Starting a Session Manager

In GroupKit, you don't run programs directly, but invoke them via another program called a session manager. The session manager is used to locate other people in your work community running GroupKit programs, and connect your programs up to them, or to start up programs of your own. Running your programs together with other people is called a conference or session. GroupKit actually comes with several different session managers, but we'll use one called the Open Registration session manager.

To start it, do the following:

- On Unix, type "open.reg &"
- On Windows, double-click "OpenReg.tcl"
- On Macintosh, double-click "OpenReg"

The first time you start up, the session manager may ask you some questions about yourself, such as your name, and store this in a preferences file It should then show the window in Figure XX (if you have problems, make sure that you have a registrar process running). The "Conferences" pane on the left shows the names of any running conferences. Selecting one will show who is in the conference in the "Participants" pane on the right. The "Conferences" menu contains a list of known GroupKit applications and lets you start up new groupware sessions.

Figure XX. Open Registration session manager.

Creating a New Conference

Assuming there are no conferences already running which we could join, let's create a new conference. We'll use a program called "Simple Sketchpad", which acts like a shared whiteboard, allowing several users to simultaneously draw freehand on a canvas.

To create it, pull down the "Conferences" menu in the session manager and select "Simple Sketchpad". A dialog box will appear, shown in Figure XX. This dialog box allows you to give your conference a name (by default it is just the name of the application) which will identify it to other users. When you've picked a name, click the "Create" button.

Figure XX. Conference naming dialog.

At this point, you'll see the name of the conference added to the "Conferences" pane in the session manager, and the Simple Sketchpad program will come up in its own window, as shown in Figure XX. You can draw on the canvas using the left mouse button. The menus let you clear the canvas, exit the program, find out about other participants in the conference, and get information about both GroupKit and the Simple Sketchpad conference. Try it!

Figure XX. Simple Sketchpad conference.

Joining an Existing Conference

We'll now have another user join the conference you've created. Find someone else on another machine nearby, and get them to start up their own copy of the open.reg session manager. They should see the conference you've created, and clicking on it will show that you are a participant in it. If you are just trying GroupKit out by yourself, create another open.reg on the same machine, and pretend you are a different person by changing your name to a new one in the entry box on the bottom.

To join the conference from their session manager, the other participant can double-click the name of the conference. This will add their name to the list of participants, and also bring up a window with their own copy of the Simple Sketchpad program. You'll also see that any drawing that was done in the first copy of the program appears in the new copy.

You'll now find that both people can draw at the same time, and that any drawings made by one user immediately appear on the screens of the other user. You'll also see a small cursor called a telepointer, which tracks the location of the other user's mouse cursor as they move around the window. More than two people can be in this conference and others can join and participate through their own session managers.

When done, you can select "Quit" or "Exit" from the "File" menu of the Simple

Sketchpad program to leave the conference. Your copy of the program will disappear, and you'll see your name removed from the list of conference participants in the session manager.

What's Really Happening Inside

To understand what is going on, lets take a step back. As you've noticed, GroupKit consists of a number of different processes, which are illustrated in Figure XX. There is a central process called the registrar, a "daemon" that should already be running on your system. Its job is to keep track of what conferences are running and who is joined to them. Each user runs a session manager, such as open.reg, which connects up to the registrar. The session managers are used to create conferences (such as our Simple Sketchpad) which again run as separate processes. As other users join conferences through their own session managers, GroupKit opens up network connections between the conference processes, so that every process in a conference has a connection to every other process in a conference.

Figure XX. GroupKit process and communications architecture.

Other Applications

There are a number of other sample applications included with the GroupKit distribution which you should try out. Some of these are illustrated in Figure XX and include:

- *Brainstorming Tool.* Allows users to brainstorm by typing in brief one-line textual ideas; all ideas appear in a listbox visible by everyone.
- *File Viewer.* Lets you load up the contents of a text file and browse through it. A multi-user scrollbar shows what parts of the file other people are looking at.
- *Hello World.* A simple program that creates a button; when pressed, the button changes on all user's screens to say hello from whoever pushed the button.
- *Text Chat.* A program similar to Unix talk, but allowing more than two participants. Text typed by each user is immediately seen by all others.
- *Tic Tac Toe.* The classic game, allowing you to play against other users.
- *Tetrominoes.* Rotate and move the different shaped polygons to get them to fit inside their containers.

Figure XX. Some example GroupKit conference applications.

Important Concepts

There are several important points about GroupKit that have been illustrated so far.

- GroupKit supports real-time distributed multi-point conferences between many users.
- GroupKit systems include both session managers for managing conferences, such as the Open Registration system, and conference applications which are the actual groupware tools, such as Simple Sketchpad.
- Every user in a GroupKit conference session runs their own copy of the conference application in a process on their own machine. These processes are connected to each other over the network.
- GroupKit is not a media-space system, which would include things like audio or video conferencing. Many of the conference applications will strongly benefit from having some kind of voice connection, such as provided by a telephone. Alternatively, if you have some sort of computer-based media space system available, it would be possible to integrate it with GroupKit, so that starting a GroupKit conference also starts the media-space system.

GroupKit User Manual. Last updated March 16, 1998 by Mark Roseman.

GroupKit User Manual - An Example GroupKit Program

Now that we've seen what GroupKit programs look like and how its runtime architecture is set up, we can start building our own conference application. There are a lot of pieces in GroupKit to explain, so what we'll do is develop a relatively sophisticated groupware program over the space of the chapter, starting with a minimalist single-user version and gradually adding features that illustrate GroupKit's programming constructs.

The program we'll build is a brainstorming tool called "Note Organizer." This tool can help a group generate and organize ideas, for example to plan a paper, software project, advertising campaign, etc. As a brainstorming tool, the program will allow users to enter ideas (a few words), each which will be represented as a text item in a Tk canvas widget. To organize the ideas, users will be able to drag them around on the canvas, grouping related ones and so on. Being groupware, everyone in the group will be able to generate and move ideas around at the same time, and see what everyone else is doing. The program will look like the one in Figure XX.

Figure XX. Screen dump of Note Organizer.

Here is how we'll approach building this example. We'll start off by creating a single-user version of the program that allows you to create ideas, but not to move them around. This will run as a normal "wish" script, without using GroupKit at all. We'll then take that program and modify it slightly so that it will run within GroupKit, before putting any code in to "share" the ideas. The next step will be to share ideas, so that when one participant creates an idea on their canvas, it appears on other participants' canvases also. We can then begin to get more sophisticated, by letting users drag ideas around the display, which will introduce some techniques that are useful when building groupware systems. Finally, we'll have the program do appropriate things when people enter and leave the session, and also add some groupware widgets that will help keep track of what people are doing when they are in the session with us. That will complete the application as illustrated in Figure XX. If you'd like to skip ahead and see what the entire program will look like, the complete listing of the final version appears at the end of this chapter.

Single-User Version

So lets start out with just a single-user version, and only worry about creating the ideas but not moving them around yet. The code - which is standard Tcl/Tk - is shown below. The program sets up some global variables to keep track of where the ideas are to be placed on the canvas, and the font to use when drawing the ideas. It then creates the canvas to hold the ideas, an entry widget to type in the ideas, and a button used to copy the idea from the entry to the canvas. When pressed, the button calls the addNewIdea proc, which gets the idea out of the entry, calls doAddIdea and clears the entry in preparation for the next idea. The doAddIdea procedure actually puts the idea into the canvas, by creating a text item at the location found in the notes(x) and notes(y) global variables. It then adjusts these variables so as to place the next note below the one just added, starting a new column when it gets far enough down the canvas.

```
set notes(x) 20
set notes(y) 20
set notes(notefont) [list helvetica 17]

proc buildWindow {} {
    frame .main
    canvas .notepad -scrollregion "0 0 800 3000" -yscrollcommand ".scroll set"
    scrollbar .scroll -command ".notepad yview"
    frame .controls
    entry .newidea
    button .enteridea -text "New Idea" -command addNewIdea
    pack .main -side top -fill both -expand yes
    pack .notepad -side left -fill both -expand yes -in .main
    pack .scroll -side right -fill y -in .main
    pack .controls -side top -fill x
    pack .newidea -side left -fill x -expand yes -in .controls
    pack .enteridea -side left -in .controls
}

proc addNewIdea {} {
    set idea [.newidea get]
    doAddIdea $idea
```

```
    .newidea delete 0 end
}

proc doAddIdea {idea} {
    global notes
    .notepad create text $notes(x) $notes(y) -text $idea -anchor nw \
        -font $notes(notefont)
    incr notes(y) 20
    if {$notes(y)>600} {
        set notes(y) 20
        incr notes(x) 100
    }
}

buildWindow
```

You should be able to run that program just fine under Tk's normal wish. Type ideas in the entry box and press the button to add them to the canvas.

Running the Example in GroupKit

The first thing you should do is add GroupKit's standard menubar, which contains menu items to exit the program, find out what other users are working on the program with you, and display an about box. Add this line before creating the main interface of your program, i.e. just before creating the canvas widget:

```
gk::menus addstandard
```

As before, we'll start our conference application using the Open Registration session manager. But first we have to tell the session manager about our new program. To do this, we need to add a line to the bottom of the GroupKit preferences file, which was created the first time you ran the session manager.

This file is in different places on different platforms:

- On Unix, it is called "preferences", and located in a directory called ".groupkit" in your home directory.
- On Macintosh, the Preferences folder in your System Folder will contain a folder called "GroupKit"; look there for the file called "preferences"
- On Windows, look for the file "preferences" in a directory called "gkprefs", stored within the directory you run GroupKit from

When adding the following line to the preferences file, you'll have to change the path from (for example) "/home/you" to be the name of the directory where you put your note organizer script file.

Note that each user has their own preferences file, so if you're running with several users, each person will have to make this change.

```
gk::preferences set prog.NoteOrganizer.src "/home/you/noteorg.tcl"  # Unix
gk::preferences set prog.NoteOrganizer.src "c:\\groupkit\\noteorg.tcl"  # Windows
gk::preferences set prog.NoteOrganizer.src "Macintosh HD:GroupKit:noteorg.tcl"  # Mac
```

Either quit and restart your session manager, or choose "Re-initialize" from the "File" menu. You should now find an item named "NoteOrganizer" under the Conferences menu. Use it to create the Note Organizer conference as before, and then join the conference from another session manager.

If you run multiple copies, you'll quickly find that if you enter ideas in one copy of the program they don't appear on remote copies. Thats because we haven't actually told the program to display ideas on all screens - GroupKit won't take care of that automatically. You will find though that items in the menubar work fine, such as the "Show Participants" item which provides information on other users in the conference.

Just One More Thing...

So now lets fix our program so when ideas are entered in one copy of the program they are sent to other users. First, quit both running copies of the program (when you quit the last one it will ask you if you want to delete the conference or keep it around; you should delete it). Now, go back into your program and change the second line in addNewIdea from:

```
doAddIdea $idea
```

to:

```
gk::to all doAddIdea $idea
```

and then re-start the program from the first session manager. After joining the conference from the second session manager, you should find that ideas entered in one copy of the program now appear in the other copy as well. That wasn't so bad!

So at this point, you've seen what is involved in getting a simple groupware program up and running in GroupKit. You've been exposed to GroupKit's session manager, learned how to tell the session manager about your new program, and how to initialize a GroupKit program. Finally, you've seen the "gk::to all" command, which is one of GroupKit's programming constructs that can make it easy to turn a single-user program into a multi-user one.

So what does the "gk::to all" do? That command arranges for the Tcl command following it (i.e. doAddIdea $idea) to be executed not only in the local program (where the idea was typed in), but also in every other copy of the program running in the conference. So, if you had not just two, but three, four or ten people joined in the conference, all of their conference processes would execute that same command. This is illustrated in Figure XX.

Figure XX. The gk_toAll command.

The "gk::to all" command is an example of a Remote Procedure Call (RPC) that GroupKit provides. GroupKit has other RPC's that differ in who the commands get sent to, as shown in the sidebar. RPC's are a fairly straightforward but effective way to turn a single-user program into a multi-user one.

You'll find when you're using RPCs that you often need to "factor" your code, splitting a routine into two parts, as you saw in the "addNewIdea" and "doAddIdea" procs. This separates the code that invokes a routine (usually called from the user interface, and which is executed only by a single user) from the code that actually performs the operation (which is executed by everyone in the conference). Not only is this a useful style to adopt for groupware, but it is good coding practice generally. Factoring your code makes it easier to change your user interface, or invoke the core of your program in entirely new ways such as from a Tcl script.

GroupKit Remote Procedure Calls

GroupKit's Remote Procedure Calls (RPCs) execute a Tcl command in the conference processes of different users in your conference session. They differ in which processes the commands are sent to.

gk::to all *cmd args*.

Execute a Tcl command on all processes in the session, including the local user.

gk::to others *cmd args*.

Execute a Tcl command on all remote processes in the session, but not the process of the local user.

gk::to *user cmd args*.

Execute a Tcl command on only a single conference process, which may be one of the remote users or the local process. The process is identified by its user's unique user number, which can be extracted from the users environment, described shortly.

GroupKit User Manual - Coordinating Multiple Users

Now that our Note Organizer can create ideas that show up on everyones' screens, lets start looking at how we can move the ideas around to organize them. To accomplish this in a single-user program, we would change the doAddIdea procedure to attach a binding to each canvas item after it is created, and reposition the item when the mouse moved, as illustrated in the following code fragment.

```
set canvasid [.notepad create text $notes(x) $notes(y) -text $idea \
                  -anchor nw -font $notes(notefont)]
.notepad bind $canvasid  "noteDragged $canvasid %x %y"

proc noteDragged {id x y} {
    .notepad coords $id [.notepad canvasx $x] [.notepad canvasy $y]
}
```

As before, we could use "gk::to all" to execute that callback in all users' programs by changing the line in noteDragged to:

```
    gk::to all .notepad coords $id [.notepad canvasx $x] [.notepad canvasy $y]
```

This will work most of the time, but is very fragile and can break if users are entering ideas very quickly. Here's why. The code above assumes that copies of an idea on different users' screens all have the same canvasid. But that is not always true.

Tk assigns canvasid's in the order in which items are created. But if two users create ideas at almost the same time, they may end up being created in a different order on each users' screens. If the first user enters "foo", the "gk::to all" adds the idea to the local canvas and then sends it across the network. If at the same time a second user enters "bar", their "gk::to all" puts it on their canvas and sends it across the network. This will result in the first user adding "foo" then "bar", while the second user adds "bar" and then "foo." The two items will be created in different orders on the different systems, and therefore the canvasid's won't match. This will be a problem when it comes to moving the items around, because we need to correctly refer to the item to move. This is shown in Figure XX.

Figure XX. Ideas created in different orders on different machines.

To make matters worse, because the doAddIdea routine (which creates the text items on the canvas) also chooses the position on the canvas, if the ideas cross paths, not only will they have different canvasid's, but they will appear in different places on different users' screens.

Problems like these are fairly common in most groupware applications. They can be solved, but only if the programmer is aware of the nuances of these situations. Lets address these problems one at a time.

Uniquely Referring to Ideas via Unique User Numbers.

The normal way you'd uniquely refer to an idea would be to give it a unique id number, such as via a counter held in a global variable that increments by one every time you have a new idea. As we saw with the canvasid's, that can create some problems in a groupware application, where the counter is not global across all processes.

What we'll do is slightly modify that scheme, so that each conference process will keep a global counter, but those counters will be unique from those of other conference processes. We will do this by prefacing each counter with a special GroupKit id number guaranteed unique for that conference process.

GroupKit generates a user number for every user in a conference process. It stores that number (and a lot of other information) in a data structure called an environment. We'll discuss environments in a moment, but for now just assume that the following code fragment will return the user's unique id number:

```
set myid [gk::users get local.usernum]
```

We can now generate a unique id for every idea, by combining our user number (held in myid) with a global counter. We can then send that combined id to doAddIdea, and tag the canvas item with the combined id. Finally we can use this unique id when we're moving the idea around. This is shown below.

```
set notes(counter) 0

proc addNewIdea {} {
    global notes
    set idea [.newidea get]
    set myid [gk::users get local.usernum]
    set id ${myid}x$notes(counter)
    incr notes(counter)
    gk::to all doAddIdea $id $idea
    .newidea delete 0 end
}

proc doAddIdea {id idea} {
    global notes
    .notepad create text $notes(x) $notes(y) -text $idea -anchor nw -font \
        $notes(notefont) -tags $id
    .notepad bind $id  "noteDragged $id %x %y"
     ...
}

proc noteDragged {id x y} {
    gk::to all .notepad coords $id [.notepad canvasx $x] [.notepad canvasy $y]
}
```

Correctly Placing Ideas

So that takes care of uniquely referring to ideas. We will now fix the problem where ideas could end up being initially placed in different locations on different screens. This problem is really being caused because the receiver (doAddIdea) and not the sender (addNewIdea) is deciding where the idea should go. The code fragment below changes this so that the sender specifies the location, although we'll still let the receiver update the position for the next idea.

```
proc addNewIdea {} {
    ...
    global notes
    gk::to all doAddIdea $id $notes(x) $notes(y) $idea
    ...
}

proc doAddIdea {id x y idea} {
    ...
    .notepad create text $x $y -text $idea -anchor nw \
        -font $notes(notefont) -tags $id
    ...
}
```

What happens now when ideas are entered at the same time by different users? If you trace through the order that messages will get executed, you'll see that both ideas will end up entered in exactly the same place, overlapping each other! When you think about it, this actually makes some sense. Imagine the real-life setting where users place real paper notes on a large board. If two people place notes at the same time, they'll both reach for the same place. Seeing their problem, they can then choose to move one of the notes somewhere else. In our computer version, users can do the same thing - see the overlapping ideas and move one of them out of the way. While its possible to build very elaborate algorithms into software to recover from situations such as these, it is often best just to let people deal naturally with these situations.

Choosing what Information to Share

So far we've tried to keep the canvases of all users completely synchronized. This is known as a "What You See Is What I See" (WYSIWIS) view, where all participants see exactly the same thing. It is also possible to have some differences between displays, creating a "relaxed-WYSIWIS" view. GroupKit supports both paradigms, but choosing which information to share and which to keep private to a user is a design decision you have to make for your own application.

To illustrate relaxed-WYSIWIS, we'll add to our program the notion of a "selected" idea, just as you'd select text in an editor or an object in a drawing program. We'll display the selected idea in a larger font. For this application, we'll decide that each participant has their own selection. Users can select an idea by clicking on it. The code that actually manipulates the selection will keep track of the selected object by adding a "selected" tag to the canvas item. Here is the code:

```
set notes(selectedfont) [list helvetica 20 bold]

proc doAddIdea {id x y idea} {
    ...
    .notepad bind $id <1> "selectNote $id"
    ...
}

proc selectNote id {
    global notes
    catch {.notepad itemconfig selected -font $notes(notefont)}
    catch {.notepad dtag selected selected}
    .notepad addtag selected withtag $id
    .notepad itemconfig $id -font $notes(selectedfont)
}
```

In this example, we made the conscious design decision to not share selection between users. However, it would of course be possible to have the selection shared, so that there is a single selection, shared among all users. This could be built by adding the appropriate "gk::to all" RPC when changing selection.

We now successfully have dealt with the problems of coordinating multiple users. Generating unique id's based on the users' unique user number let us refer to objects uniquely, while letting the sender decide on the idea's position resulted in ideas appearing in the correct location on all screens. Finally, we showed that it may be desireable to have some information, such as selection, that is not shared between all users.

GroupKit User Manual - Using Environments to Find out Information about Users

At this point we can correctly create and move ideas, with the ideas appearing in the correct place on every user's screen. Lets now extend the program to identify who entered each idea. GroupKit keeps track of a designated color for each user which we can use to do this (participants can specify their personal color in their preferences file or via the "Pick Color" menu item in their session manager).

GroupKit stores each user's color (as well as other information) in a data structure called an environment. Environments are structured as a tree, where each node can contain either other nodes or a value. A node is referred to by its position in the tree, with each level of the tree separated by a dot. So for example, local.usernum refers to the node usernum which is a child of local which is a child of the root node of the environment. This is similar to how Tk refers to windows in the window hierarchy.

Information about each user is stored in the environment called gk::users, stored underneath a key holding the user's number. The special key "local" is an alias pointing to the local user's information. So for example if the local usernum (retrieved as "gk::users get local.usernum") was 20, you could access the local user's color as either "gk::users get local.color" or "gk::users get 20.color".

Information on other user's is stored under their respective user numbers. You can get a list of all the user numbers of remote users via the special alias "gk::users keys remote". More examples of how to use environments are described in the sidebar "Using Environments".

Figure XX. Users environment.

Using Environments

Environments have a number of different commands used to manipulate them. If you're familiar with Extended Tcl's keyed lists, you'll find some similarities. Some of the commands include:

1. gk::environment envname. Create a new environment, which also creates a Tcl command called "envname" used to access the environment.
2. envname set key value. Set the value of the node located at key.
3. envname get key. Get the value of the node located at key.
4. envname keys key. List the children of the node located at key.
5. envname delete key. Delete the node located at key (and all nodes below it).
6. envname exists key. Check if a node located at key exists in the environment.
7. envname destroy. Destroy an environment, all data in it, and its Tcl command.

GroupKit maintains a number of environments internally. You've seen the gk::users environment which holds information about all the users in the conference, including their color (the key "color"), their full name ("username"), their userid ("userid"), and host ("host"). You can also store your own information in the users environment, to be used by your application. The other environment you've encountered briefly is the gk::preferences environment, which is used within the preferences file to specify where different conferences exist. Environments are also used extensively throughout the internals in GroupKit.

Lets add this into our program now. When we create an idea, we'll send our user number along with the idea. When creating the text item to display the idea, we'll change its color to be that of the user who created it. We'll also add a canvas tag to the text item identifying the user who created it, which we'll come back to later. Here are the changes in the code:

```
proc addNewIdea {} {
    ...
    gk::to all doAddIdea $id [gk::users get local.usernum] $notes(x) $notes(y) $idea
    ...
}

proc doAddIdea {id usernum x y idea} {
    ...
    set color [users get $usernum.color]
    .notepad create text $x $y -text $idea -anchor nw -font  $notes(notefont) \
```

```
        -fill $color -tags [list $id user$usernum]
    ...
}
```

Environments provide a convenient way to find out information on other users. They also have other features that can help in building groupware, which we'll return to later. In the meantime however, we'll look at another GroupKit feature called conference events.

GroupKit User Manual. Last updated March 16, 1998 by Mark Roseman.

GroupKit User Manual - Conference Events

As you've noticed, GroupKit automatically takes care of users joining and leaving conferences, without your intervention. However, its often useful to be notified when people come and go. To do this, GroupKit provides a set of three conference events.

Conference events are similar to bindings that you can attach to widgets. But rather than executing a piece of callback code when something happens to a widget (e.g. the user presses a button), conference events execute a piece of code you provide as users join and leave the conference.

To specify handlers for conference events, you use the "gk::event bind" command, which works very similarly to Tk's bind command, right down to its use of "percent substitutions" to pass event parameters to your callback code. The general format of the command is:

```
gk::event bind event-type callback-code
```

New Users

The first of the conference events is the newUserArrived event, which signifies a new user has just joined the conference. We can extend our program to watch for new users arriving and displaying a dialog box with the code below. When this event is generated, information about the new user has already been stored in the users environment which was described above. One of the pieces of information available besides color is the user's name, so we'll display that in our dialog box. The percent substitution "%U" refers to the user's unique user number, a parameter of this conference event. Here, we'll embed the callback code directly in the gk::event bind command. As with Tk bindings, we could also have the binding call a separate procedure to handle the event.

```
gk::event bind newUserArrived {
    set newuser %U
    set name [gk::users get $newuser.username]
    toplevel .new$newuser
    pack [label .new$newuser.lbl -text "$name just arrived."]
    pack [button .new$newuser.ok -text Ok -command "destroy .new$newuser"]
}
```

Users Leaving

The second conference event is generated when a user leaves the conference. Again, the %U refers to the user's unique user number. The information for this user in the gk::users environment is still available at the time this event is generated. In our program, we may choose to respond to this event by changing the color of any ideas generated by the leaving user to black, as illustrated in the code below (remember that we tagged all ideas created by that user with the word "user" appended with their user number).

```
gk::event bind userDeleted {
    catch {
        .notepad itemconfig user%U -fill black
    }
}
```

Updating Latecomers and Saving Conference Results

The final conference event generated by GroupKit is the updateEntrant event, used to update latecomers to a conference already in progress. Unlike the previous two events, GroupKit sends this event to only one of the existing conference processes, chosen arbitrarily. The process receiving this event should respond to it by sending whatever information is necessary to the latecomer to bring them up to date. In our program, this would mean sending them a copy of all of the ideas that have been generated so far.

Figure XX. Updating latecomers.

This information is normally sent via the "gk::to *user*" RPC, described earlier. This call needs the user number to send the message to, again extracted from the event via the "%U" substitution.

```
gk::event bind updateEntrant {
    foreach item [.notepad find all]
        set x [lindex [.notepad coords $item] 0]
        set y [lindex [.notepad coords $item] 1]
        foreach tag [.notepad gettags $item] {
            if {[string range $tag 0 3]=="user"} {
                set usernum [string range $tag 4 end]
            } else {
                if {$tag!="selected"} {set id $tag}
            }
        }
        set idea [.notepad itemcget $item -text]
        gk::to %U doAddIdea $id $usernum $x $y $idea
    }
}
```

The updateEntrant event is also used to make conferences persist (that is, make them stick around after the last user has left, so that they can be rejoined later). You've noticed that when you quit the last program in the conference, you're asked if you'd like to delete the conference or have it save its contents (persist). If you ask it to persist, GroupKit sends an updateEntrant event to your program, but rather than asking it (via the %U parameter) to update a new user, GroupKit passes a special user number that causes your program to send messages to a special persistence server that records them. When a user next joins the conference, the messages stored in the server are played back, exactly as if they were sent from the last user.

It is also possible to create your own events, specific to your application. This can be useful as your programs get large, as a way of communicating changes between different parts of your program. See the gk::event and gk::notifier pages in the reference manual for more information.

GroupKit User Manual - Groupware Widgets and Awareness

Our program is starting to get quite sophisticated, allowing ideas to be created and moved around on multiple screens, displaying different users' ideas in different colors, responding to new and leaving users, updating latecomers to the conference, and even persisting when all users have left the conference.

As a conference participant, you may sometimes find it hard to follow when several people are working at the same time. Ideas are being rapidly moved around, and its unclear who's doing what when. Also, because the canvas is quite tall, it can become hard to track where people are working.

GroupKit provides a number of different awareness widgets to help with these sorts of problems. Keeping track of where people are working and what they are doing is a common problem in a number of different groupware programs, so our widgets can be easily added to many different applications.

Telepointers

You've probably noticed yourself that when people get together around a whiteboard to discuss some sort of problem, that along with a lot of drawing, there's also a lot of pointing or gesturing to objects drawn on the whiteboard as well. In fact, some studies of how people use drawing surfaces like whiteboards found that gesturing is actually more common than drawing. GroupKit's telepointers provide a way to communicate this important gesturing information in groupware applications.

Telepointers, also known as multiple cursors, are used to provide very fine-grained information about where other users are working in the application. As other users move around the application, you'll see a small cursor which follows their mouse cursor. Just tracking those cursors - which you're usually just peripherally aware of - provides a surprising amount of information about who's doing what in an application, what objects people are working with, and how active different people are (Figure XX). Telepointers can be attached to any Tk widget.

Figure XX. GroupKit's telepointers.

Adding telepointers to our example is quite straightforward, requiring only one line of code.

```
gk::telepointers attach .notepad
```

Multi-user Scrollbars

While telepointers are good at answering questions like "what are people who are working close to me doing exactly?", multi-user scrollbars answer questions like "where are other people working in this large document?" They provide a more coarse-grained sense of awareness.

Figure XX illustrates GroupKit's multi-user scrollbars. They consist of two parts: a conventional Tk scrollbar on the right, and a set of bars showing where users are located in a document to its left. The conventional scrollbar is used normally, to scroll your own view in the document. As you scroll, you'll also notice one of the bars to the left scrolling with you. That bar is showing your position in the document. As other users scroll their own views, the other bars on their display will change to show their new positions. Clicking on each bar will display a popup reminding you which user the bar is associated with. Multi-user scrollbars allow you to quickly find where others are working in a large document and - by aligning your view with theirs - join them.

Figure XX. Multi-user scrollbars.

Multi-user scrollbars are added to applications in exactly the same way that conventional scrollbars are. In fact, the only difference is that instead of using the Tk scrollbar command, you use the GroupKit gk::scrollbar command.

```
proc buildWindow {} {
    ...
    gk::scrollbar .scroll -command ".notepad yview"
    ...
}
```

GroupKit also comes with a number of other widgets. You've already seen the menubar, which includes items such as an Exit command, an about box for GroupKit, and an item which brings up a dialog box giving you information about the different users in the conference with you. There's also a widget called the "radar", which is similar to the multi-user scrollbar, but augments location information with a two-dimensional miniature of the entire document.There's also a widget that is included to make it easier to build online help into your application.

At this point we've completed our Note Organizer program; you'll find the complete program listing at the end of this chapter. The next section will go on to talk about a very different approach to building groupware applications with GroupKit.

GroupKit User Manual - Shared Environments

When writing the Note Organizer application, we've used a very direct style of programming; when we wanted to move an idea around, we literally sent a message to the other users' canvas widgets to move the idea. This approach works very well for small, highly graphical applications, but doesn't always scale up well to larger applications, particularly if you'd like to allow different users to customize how they work with their applications. This section discusses an alternative approach to programming in GroupKit, using some features of environments.

To understand how environments can be used to build groupware programs, we'll have to first describe a paradigm called "Model-View-Controller" (MVC), which was first introduced in Smalltalk as a way to construct (single-user) user interfaces. The MVC paradigm suggests that an application should be divided into three distinct components:

1. A "model" represents the underlying data structure that is to be displayed.
2. A "view" looks at the contents of the model and creates an on-screen visual representation of it.
3. A "controller" reacts to user input events and sends changes to the model.

Under this paradigm, the controller never changes the onscreen view directly. Instead it changes the model, and the changes to the model are picked up by the view which does cause a change onscreen. This is illustrated in Figure XX.

Figure XX. Model-View-Controller paradigm.

This approach can be a bit hard to follow at first, since input events only indirectly affect what is displayed onscreen, but has a number of advantages. First, by clearly separating out your underlying data from the view of that data and how it is manipulated, it encourages you to build a very clear data model, which can help you understand exactly what it is your program is manipulating. By doing this separation, you also find it easier to modularize your code, rather than trying to do everything in one place. Finally, the paradigm makes it easier to have multiple views on the same data structure, for example allowing you to view a table of numbers as either a spreadsheet or a graph.

So how does this relate back to GroupKit? Recall that we can use GroupKit's environments to store a fairly arbitrary set of data (such as names and colours in the gk::users environment), arranged in a tree structure, where we can associate a string value with any node in the tree. This can serve as an underlying data structure for a wide variety of applications, and so is an appropriate choice for our Model under the MVC paradigm. Environments can also generate events (similar to the conference events used for announcing new users etc.) when they are changed. These events can be received by other portions of your program, such as a View which updates the screen to match the change in the Model. Finally, parts of your program that handle user input (the Controller portion) can react to these changes by modifying the environment.

We've taken environments one step further, and allowed them to be shared between all the users in a conference. When any user makes a change to their own copy of the environment, that change is automatically propagated to all other users' copies. So while a user's input event may change their own environment, which generates an event to update their onscreen view, the change also gets sent to every other user's environment, generating the same event and updating their view - instant groupware.

These shared environments and the MVC paradigm really shine on larger applications, but here we'll just present a toy example to illustrate the mechanisms. Our data model will be held in an environment called stocks and consist of a single piece of information, the value of the fictional GroupKit Inc. stock. Our view will consist of an onscreen label showing the value, and an entry widget will allow us to change the value.

After initializing GroupKit, we begin by creating the stocks environment, using the gk_newenv command. The -peer flag tells GroupKit to create this as an environment shared with all its peers. We'll then give our GroupKit stock an initial value.

```
gk::menus addstandard
gk::environment -peer stocks
stocks set groupkit 1
```

Next, we'll create our view, using a label widget. The "stocks bind changeEnvInfo" attaches a binding to the environment, so that whenever a change is made to it, the following code will be executed. There are also events for when a piece of information in the environment is first added (addEnvInfo), and when an item is removed (deleteEnvInfo)* . The event parameter %K refers to the name of the key in the environment that changed; we could have omitted they key here since there is only one key in the environment.

```
label .view -text "GroupKit Inc. value is [stocks get groupkit]"
pack .view -side top
stocks bind changeEnvInfo {
    if {%K=="groupkit"} {
        .view config -text "GroupKit Inc. value is [stocks get groupkit]"
```

```
    }
}
```

Finally, we create the controller, using an entry widget. When we hit the "Return" key in the entry, its value is retrieved and stored in the stocks environment. The view code should then notice the change and update the display. Finally, because it is a shared environment, the change will show up in all other users' environments, updating their views as well.

```
pack [entry .controller] -side top
bind .controller  {stocks set groupkit [.controller get]}
```

We could easily replace either the controller or view with ones that behaved differently, without changing any other parts of the program (or they could even display several views, since multiple bindings on an event are allowed). Different users could also use different versions of the programs in the same conference. This works because only the models (the shared environment stocks) of different users directly communicate with each other, and not the views or controllers. So for example here is a view that uses a canvas widget to display a horizontal bar whose length depends on the value of the stock held in the environment:

```
pack [canvas .barview] -side top
.barview create rectangle 0 0 [expr [stocks get groupkit]*5] 10 -anchor nw -tags bar
model bind changeEnvInfo {
    if {%K=="groupkit"} {
        .barview coords bar 0 0 [expr [stocks get groupkit]*5] 10
    }
}
```

In summary, GroupKit's shared environments can serve as an easy way to implement the Model-View-Controller paradigm, and extend it to support multi-user applications. Environments can take care of the housekeeping chores in updating and synchronizing multiple copies of a data structure (such as the multiple copies of the canvas widget used in the Note Organizer - though the canvas data structure had the advantage of having its view built-in!), and can help to modularize your application. Its a judgement call whether to use shared environments or the more direct RPC's in your application (they can be mixed for different parts of course). In general, if your application becomes larger, if you're dealing with information that can be changed in various ways by your program, or if it can be viewed in different ways, then environments are usually an appropriate choice.

---

gk::to

Multicast remote procedure calls

Synopsis

gk::to ?-serialize? all *cmd args...*
gk::to others *cmd ?arg...?*
gk::to *userlist cmd ?arg...?*
gk::to configure *option value*

Description

The gk::to command is used to send multicast remote procedure calls (RPCs) between GroupKit processes. The options on this command determine who the RPCs are sent to. You can specify a list of one or more GroupKit user numbers to refer to the processes, use the keyword "all" to refer to all processes in the current conference including the local process, or the keyword "others" to refer to all processes in the current conference except the local user.

The flag "-serialize" may optionally be added to the "all" form of this command, to guarantee that messages arrive at all processes in the same order, even if they were sent from different processes in the conference. This is accomplished by serializing the messages through a single conference process, which can add some delays. Without this flag, there is no guarantee of message ordering across processes, except that messages sent from a single site will always arrive at their destination in the same order they are sent.

The body of an RPC is an arbitrary Tcl command. The execution environment for this command is completely unrestricted; there are obvious security implications of this!

The "configure" option allows you to configure how the gk::to command behaves. Currently, two options are available: "environment" and "method". The former is used to specify the environment which is used to send RPCs. In GroupKit, all RPCs are routed through environments; this option is set by the run-time environment, so you should normally not need to change it.

The second option, "method" allows you to override how GroupKit actually sends the RPC messages. The default method (general) sends

them through the environment's router. The other alternative, "direct" can only be used with peer environments, and by-passes the environment's general routing mechanism and sends directly to the other peer processes, which results in a substantial speed-up.

---

gk::event

Generate and respond to local events

Synopsis

gk::event notify *event params*
gk::event bind *event script*
gk::event delete *binding*

Description

The gk::event command provides a way to have one part of the application generate an event that another part of the application can respond to, without either part having to know directly about the other.

Events consist of an event name, and a list of parameters providing more details about the event. Each parameter is itself a two item list, consisting of a key (which must be a single character) and an associated value. An event is generated using the "notify" option.

To respond to an event, a binding must be established. The "bind" option arranges for *script* to be called whenever an event named *event* is generated. The "bind" option returns a binding, which can be passed to the "delete" option if your program no longer wishes to be notified of the event. The event name specified in the "bind" option can also include wildcards, following the rules of the "string match" command.

To get the value of event parameters, the script undergoes "percent substitution" before being evaluated. Occurences of the two characters percent ("%") followed by the character representing an event parameter are replaced by the value of that event parameter. This is the same mechanism used to access parameters in Tk's event mechanism.

Note that the gk::event command uses a single instance of a gk::notifier object. You can create other instances of gk::notifier objects for other uses in your application.

Example

```
set binding [gk::event bind newUserArrived "handleNewUser %U"]
...
gk::event notify newUserArrived [list [list U 25]]
...
gk::event delete $binding
```

---

gk::schedule

Manage the execution of repetitive commands

Synopsis

gk::schedule *tag command ?timer?*
gk::schedule cancel *tag*

Description

The gk::schedule allows you to deal with executing repetitive commands so that commands do not cause a backlog. Tcl commands are added to a queue without duplicates, so that only a single Tcl command will be executed within a given time interval.

For example, lets say that in response to a received "move object" command, you would like to do an "update idletasks" after moving the object, so that the screen is updated. If you received thirty move object commands a second, you wouldn't be able to update the screen fast enough to process the messages, and they would be queued up. With gk::schedule, you could choose to process the messages right away, but only update the screen a maximum of ten times a second.

To schedule a command, you need to assign it a tag (all commands you wish to treat as part of the same batch, such as screen updates, should be given the same tag). You then specify the Tcl command to execute, and optionally a timer value (the default is 100 milliseconds), controlling how often commands will be executed. To cancel a scheduled command, if it has not already been executed, use the "cancel" option.

Example

```
proc moveObject {id coords} {
    # move the object...
    ...
    # update the screen a maximum of five times a second
    gk::schedule screenupdate "update idletasks" 200
}
```

---

gk::environment

Shared tree-based data structure

Synopsis

gk::environment *?options? envName*

Description

The gk::environment command creates a new instance of an environment, a tree-based data structure that can optionally be shared between GroupKit processes. Environments are pervasive in GroupKit, and are used to store a wide variety of data.

Environments store data in a tree structure. Nodes in the tree are referred to by a key, using a period as a hierarchy delimeter. For example "users.3.color" refers to a node called "color" which is a child of a node called "3" which is a child of a node called "users" which is a child of the root of the tree. Any node can contain zero or more children. One feature of environments is that nodes can contain either data or children, but not both.

Three options can be specified when the environment is created if the environment is to be shared. Specifying "-server" creates a centralized environment that other processes can connect to. The "-client" option is used to connect to a previously created centralized environment. Finally, the "-peer" option can be used to specify a replicated sharing strategy, commonly used in GroupKit conference processes.

Environments also generate events when their contents changes; these events are similar to those implemented by the gk::event command. As well, there is a mechanism to extend the behavior of environments by adding new operations or redefining existing ones.

Environment Command

The gk::environment command creates a new Tcl command named *envName*, which is used to work with the environment. It supports a number of different operations:

envName set *key value*

Place the given *value* into the node *key*. If the node does not exist, it is created, along with all intermediate nodes. If the node previously stored data, that data is replaced. If the node previously had any children, all child nodes are deleted.

envName get *key*

Return the value associated with the node. If the node does not exist, an empty string is returned. If the node does not hold data but instead has child nodes, a complex representation of all the children and their data (based on Extended Tcl's keyed lists) is returned.

envName exists *key*

Returns 1 if the given node exists in the environment, 0 otherwise.

envName delete *key*

Delete a particular node in the environment, and any children of that node.

envName keys *?key?*

Return a list of all the children of the given node. If the node does not exist or has no children, the empty list is returned. If *key* is omitted the list of all children of the root of the environment are returned.

envName destroy

Destroy the environment data structure and its associated Tcl command.

envName bind *event script*

Create a new event binding such that when the environment generates *event*, *script* will be executed. This command returns an event binding id. See the section on "Event Bindings" for details on the events that are generated and their parameters.

envName delbind *binding*

Delete a previously created event binding.

envName export

Dump the entire contents of the environment into a single string that can be passed to the "import" command.

envName import *data*

Load the contents of the environment from a string which was generated by the "export" command.

envName debug *key*

Provide a reader-friendly version of the environment's contents.

envName option *operation ?args?*

Environments actually support two complete hiearchies, the normal "data" hierarchy, and a second "option" hierarchy, which is typically used to store information controlling how the environment behaves. The "option" command is used to access this subtree. Available operations (which parallel those in the data hierarchy are):

envName option set *key value*
envName option get *key*
envName option delete *key*
envName option exists *key*
envName option keys *?key?*

envName command *operation ?args?*

When an environment command (e.g. "envName set") is invoked, the particular operation ("set") is looked up in an internal table to determine how to perform that operation. Using the "command" operation of environments, it is possible to manipulate that table, thereby adding new commands to the environment or replacing existing ones. Handlers for these operations are either C built-ins or Tcl procedures; the latter are called with the name of the environment, the operation, and any additional parameters as arguments. Additionally, a list of "routings" for shared environments can be specified.

envName command set *subcmd script*

Specify *script* as the handler for the environment operation specified by *subcmd*.

envName command get *subcmd*

Return the name of the handler for *subcmd*. This will either be the name of the Tcl procedure, or "<builtin>" for C handlers.

envName command delete *subcmd*

Delete the operation *subcmd* from the environment.

envName command list

Return a list of all known subcommands of the environment.

envName command rename *subcmd newsubcmd*

Rename the given *subcmd* so that it is now referred to by the name *newsubcmd*.

envName command routing *subcmd ?routings?*

Specify a list of routings for this command, or return the current list of routings. Routings are used by shared environments to control how environment messages are distributed among processes.

envName addDependent *key process ?type?*

For shared environments, the "addDependent" command provides a mechanism to specify that a particular node in the environment is dependent on the existence of a GroupKit process, and that when that process goes away, the corresponding node should be deleted. The optional *type* specifies whether the node is part of the "data" hierarchy, or the "option" hierarchy.

Event Bindings

Environments generate events when nodes are changed. These are similar to those generated by the gk::event command, but environment events are intercepted by the *bind* environment command.

When a node is added to the environment, an "addEnvInfo" event is generated. Its parameter is "%K" specifying the key of the node that is added. When an existing node is changed, a "changeEnvInfo" event is generated, again with a "%K" parameter. When a node is deleted, a "deleteEnvInfo" is generated, again with a "%K".

As well, a second event is generated, named after the node affected. For example if you add a node "users.3.color", an event named "users.3.color" will be generated. This allows you to bind on particular nodes. Remember also that event bindings support wildcards, so you could bind on "users.*.color". This feature lets you design bindings without worrying about a lot of parsing code to determine if you have the right event. For these types of events, parameters are "%K" (the key), "%V" (the value at the key), "%P" (the previous value), "%O" (the operation; one of addEnvInfo, changeEnvInfo or deleteEnvInfo) and "%1" through "%9" for each dot separated component of the key (e.g. "%3" would be "color" in the example).

---

gk::users

Keep track of information on conference users

Synopsis

gk::users get *key*
gk::users keys *?key?*

Description

The gk::users command is used to retrieve information about the current users of a conference, or those currently connected to a GroupKit session manager.

Each user has subtree underneath the root of the environment, named after their unique user number. Typical information stored under that node would be "color", "username", and optionally information such as "title", "company", "email", and "www" as displayed by the business card widget.

Several "aliases" are available. If rather than a unique user number the toplevel node is specified as "local", it will be mapped to the local user's unique user number. In particular, you can retrieve this number via the key "local.usernum". Specifying "gk::users keys remote" is a shorthand for returning the list of all unique user numbers, except for that of the local process.

Supported environments commands include "get" and "keys". Despite looking like an environment, the gk::users command actually retrieves its information from the registry, a separate environment.

---

gk::conference

Miscellaneous conference facilities

Synopsis

gk::conference originator
gk::conference run

Description

The gk::conference command provides various utilities needed to support GroupKit conference applications (as opposed to session managers, the registrar, or other GroupKit processes).

The main routine you will use is "gk::conference originator", which will return "1" if the current process first initiated the conference, and "0" otherwise. You can use this for example so that only the conference originator will preload various data structures.

The other routines is used internally by other parts of GroupKit. The "gk::conference run" command is invoked when the conference process is first created (see confStarter.tcl) to start the conference running.

---

gk::sessionmgr

Utilities to build session manager applications

Synopsis

gk::sessionmgr init
gk::sessionmgr run *prog*
gk::sessionmgr create *confname conftype*
gk::sessionmgr join *id*

Description

The gk::sessionmgr command provides various utilities needed to support the development of GroupKit session management applications (as opposed to conferences, the registrar, or other GroupKit processes). Primarily, this module maintains a set of runtime environments and generates a series of events, described under "Runtime", below.

The "gk::sessionmgr init" command initializes the session manager runtime. Normally, you do not need to invoke this directly, but can instead rely on the "gk::sessionmgr run" command, described momentarily. This routine will prompt the user for a registrar to connect to, establish the necessary connections to its registrar and environments, create the launcher server to spawn new conferences, and arrange for events to be generated. All of this is described in the section "Runtime".

The "gk::sessionmgr run" command is used to run a particular Tcl script as a session manager, initializing the session management facilities if this has not already been done. This is normally the routine invoked for example by shell scripts such as "open.reg".

To create a new conference, you can call the "gk::sessionmgr create" conference. While the name you specify for the conference is arbitrary, the type should designate one of the conference types specified in the GroupKit preferences file. If successful, this routine will generate a <CONF_CREATED> event (see below) and return the process id for the newly created conference process.

To join an existing conference, call the "gk::sessionmgr join" command, specifying the id of the conference (not of a specific process). If successful, this routine will generate a <CONF_JOINED> event (see below) and return the process id for the newly created conference process.

Runtime

There are several environments maintained by the gk::sessionmgr command, as well as several different events that are generated. You will need to interact with these environments and respond to these events to implement your own session manager applications.

The "gk::confs" environment maintains a list of all the current conferences. All of these are stored under the "c" key, and represented by their unique conference id number. Therefore, if you need to retrieve a list of all known conference id numbers, you can call "gk::confs keys c".

Each conference listed in the gk::confs environment also has its own environment, which maintains information about the conference name, type, and list of users. This environment is called "gk::confs_$confid", where "$confid" represents the unique conference id number. You can obtain the conference name via the "name" key, and its type via the "type" key. The list of users in the conference is stored (via unique id number for the user's conference process), under the "users" key.

When conferences or users are added or deleted, events are generated, which your application can capture via the "gk::event bind" command. The events to look for are <CONF_ADDCONF>, <CONF_DELCONF>, <CONF_ADDUSER> and <CONF_DELUSER>. All four of these events use the key "C" to hold the unique conference id number for the conference effected, while the latter two events also use the key "U" to hold the id for the user's conference process.

Two additonal events are generated for when the local session manager successfully creates a new conference or joins an existing one. These are the <CONF_CREATED> and <CONF_JOINED> events. Both of these use the key "C" to correspond to the conference unique id number, and "U" for the id of the user's process.

---

gk::preferences

Manage user preference information

Synopsis

gk::preferences *environment-command ?args?*
gk::preferences read

Description

The gk::preferences environment is a normal GroupKit environment, used to hold various preference information. Information is normally loaded from the GroupKit preferences file.

Besides normal environment commands (e.g. set, get, keys), the "read" command will cause the environment to be loaded from the preferences file. The gk::users command is used to retrieve information about the current users of a conference, or those currently connected to a GroupKit session manager.

Each user has subtree underneath the root of the environment, named after their unique user number. Typical information stored under

---

gk::telepointers

Multiple cursors

Synopsis

gk::telepointers attach *widget*

Description

The gk::telepointers command allows you to add multiple cursors to a Tk widget and all of its children. Telepointers are small cursors that represent the mouse cursor of other users on your own screen, allowing you to keep aware of their work.

Use "gk::telepointers attach" to attach telepointers to a particular widget. All of your mouse movements will then be automatically sent to all other users so they can display your telepointer.

GroupKit Reference Manual. Last updated March 30, 1998 by Mark Roseman.

gk::scrollbar

Multi-user scrollbar

Synopsis

gk::scrollbar *?scrollbar-options? pathName*

Description

The multi-user scrollbar is a groupware version of a conventional Tk scrollbar. Alongside the normal scrollbar is a thin canvas which displays a set of rectangles, one for each user in the conference. These rectangles track the position of each user's own scrollbar though the document, providing a sense of awareness of what areas of the document others are scrolled to.

The gk::scrollbar command provides a drop-in replacement to Tk's scrollbar command. It accepts the same set of configuration options, and the widget command is used in the same way.

GroupKit Reference Manual. Last updated April 6, 1998 by Mark Roseman.

gk::radar

Radar view awareness display

Synopsis

gk::radar *?-document document? pathName*

Description

The radar view widget is a passive awareness display that will track the movements of users in a two-dimensional document. The radar responds to changes in GroupKit's awareness model to update its display. Typically, the radar view will also be used in conjunction with GroupKit's scrollbars, which generate changes in the awareness model. This is seen in the following code fragment:

```
grid [canvas .c -scrollregion "0 0 2000 2000"] \
    -column 0 -row 0 -sticky nwes
grid [gk::scrollbar .hs -orient horizontal -command ".c xview" \
    -multiuser no] -column 0 -row 1 -sticky we
grid [gk::scrollbar .vs -command ".c yview" -multiuser no] \
    -column 1 -row 0 -rowspan 2 -sticky ns
grid [gk::radar .r] -column 2 -row 0
```

---

gk::menus

Standard menubar facilities for GroupKit applications

Synopsis

gk::menus addstandard *?pathname? ?parent?*
gk::menus addhelp *name html ?menubar?*
gk::menus addconferences *?menubar?*

Description

The gk::menus command provides some convenience routines allowing you to quickly and easily add support for some common menu items to your GroupKit application.

The "gk::menus addstandard" command adds a standard menubar to (by default) the root window of your application, containing items for online help, displaying the list of participants, exiting, debugging, and so on.

The "gk::menus addhelp" command allows you to add a menu item to the help menu, which when invoked will display a help message that you specify (formatted as HTML) in a dialog box.

Finally, the "gk::menus addconferences" command can be used by session managers to add a list of known conference types to the menubar. When a conference is chosen, a dialog box prompts for the name of the conference, and then "gk::sessionmgr create" is called to create the new conference.

GroupKit Reference Manual. Last updated March 30, 1998 by Mark Roseman.

gk::participants

Various facilities for interacting with user information

Synopsis

gk::participants show
gk::participants card *usernum*
gk::participants email *usernum*
gk::participants webpage *usernum*
gk::participants note *usernum ?contents?*
gk::participants portrait *usernum*
gk::participants sendportrait *usernum*
gk::participants receiveportrait *usernum portrait*

Description

The gk::participants command provides a set of routines for graphically dealing with the information that GroupKit provides for each user.

The "gk::participants show" command brings up a participants dialog, showing all the users. Popup menus on each user allow for further operations.

The "gk::participants card" routine brings up a business card dialog window for a particular user, providing detailed information not shown in the participants dialog.

The "gk::participants email" routine calls an external email program to send a message to the indicated user, i.e. by bringing up a mail compose window and filling in the "To" field. Currently this only works on Unix with exmh.

The "gk::participants webpage" routine will display a user's web page if available, using an external web browser. Currently this only works on Unix with Netscape.

The "gk::participants note" command can be used to send a "sticky note" to another user. If the contents of the note are specified, the note is sent automatically, otherwise the local user is prompted for the contents before sending.

The "gk::participants portrait" command returns the image name of the specified user's portrait. This can then be used as the "-image" parameter

for a widget. If the portrait is not currently available, this command returns the empty string, and initiates a query for the portrait. When it is received, a "portraitReceived" event is generated, which can be trapped with "gk::event bind". The fields are %U for the usernum, and %P for the name of the portrait image. The "sendportrait" and "receiveportrait" commands are used internally to ship portrait images over the network.

GroupKit Reference Manual. Last updated June 28, 1998 by Mark Roseman.

gk::rpc

Maintain socket connections between processes

Synopsis

gk::rpc listen ?-port *port*?
gk::rpc connect *host port*
gk::rpc send ?-class *class*? *sock message*
gk::rpc atclose *sock cmd*
gk::rpc active
gk::rpc hostport *sock*
gk::rpc handler *sock class* ?*cmd*?
gk::rpc handler default ?*cmd*?

Description

The gk::rpc command is used internally within GroupKit, and provides a higher-level interface on top of the standard Tcl socket facilities. In particular, the gk::rpc command provides for bi-directional channels over which Tcl commands can be sent, reuses existing socket connections where possible, and allows flexible handling of received messages.

The "gk::rpc listen" command is used to setup a listening socket that other processes can connect to; this must be the first gk::rpc command invoked. You can optionally specify a port to listen on, otherwise an unused port is selected. This command returns the port that the socket is listening on.

The "gk::rpc connect" command opens a socket connection to another process that has previously set up a listener with "gk::rpc listen". A socket identifier is returned that can be used to send messages over the socket. If an existing connection to the external process has already been made, a new socket is not created but the existing identifier is returned.

The "gk::rpc send" command sends a message over a socket. The optional "class" argument can be used to specify the type of message, which the handler mechanism (described below) can take advantage of. The message itself is a single string, typically containing a Tcl command to be evaluated.

The "gk::rpc atclose" command arranges for a particular Tcl command to be executed when a particular socket is closed.

During the scope of handling a received message, the "gk::rpc active" command will return the socket identifier from which the message was sent. When no message is currently being handled, this command returns an error.

The "gk::rpc hostport" command returns a two item list, containing the host and port of the indicated socket. Note that the port is actually the port of the remote processes' listener, rather than the actual port for this connection.

The "gk::rpc handler" mechanism allows you to define how incoming messages are handled. By default, all incoming messages are evaluated as Tcl commands. You may instead choose to install your own routine, which will be called whenever a message of the indicated class comes in on a given socket. The "default" option lets you provide a global handler, which is a fallback for when a specific handler for a class and socket is not provided. Omitting the "cmd" parameter on either of these forms will return the current handler.

---

gk::debug

Debugging facilities

Synopsis

gk::debug console
gk::debug message *msg*
gk::debug warn *msg*
gk::debug error *msg*
gk::debug show *type flag*
gk::debug timing reset
gk::debug timing checkpoint *description*
gk::debug timing dump

Description

The gk::debug command provides a variety of debugging and profiling utilities.

The "console" command brings up a debugging console window, modeled after a Smalltalk workspace. This allows interactive access to the application's Tcl interpreter.

The "message", "warn" and "error" commands are used to display various types of messages on standard output. The "show" command is used to control whether or not messages of each type are ignored (the default) or displayed. Note that on Windows and Macintosh, you will need to do a "console show" to see these messages.

Finally, the "timing" command provides a mechanism for doing simple profiling. To profile a block of code, first call "timing reset", and then call "timing checkpoint" at any number of positions throughout the code. When done, call "timing dump" which will output a table containing the amount of time between each checkpoint.

---

gk::registry

Convenience routines for accessing the global registry

Synopsis

gk::registry ?-registry *registry*? addprocess
gk::registry ?-registry *registry*? addgroup
gk::registry ?-registry *registry*? addgroupmember
gk::registry ?-registry *registry*? addenvironment *env type*

Description

The gk::registry command provides a simplified front end for storing information in the default system registry, or optionally, any other registry.

The "gk::registry addprocess" command adds a new entry to the registry's directory containing information about the local process. In particular, the process' host, port, usernum and group are stored.

The "gk::registry addgroup" command adds a new entry to the registry's directory containing information about the group the local process belongs to. The group represents the entire collection of processes started by the user (e.g. their session manager and all created conferences). The local process is added to the new group entry.

The "gk::registry addgroupmember" command adds the local process to an already existing group in the registry's directory.

The "gk::registry addenvironment" command publishes information about the specified environment into the registry. The type of the environment is stored, and the local process is registered as a maintainer of the environment, meaning others can connect to the local process and thereby connect to the environment.

---

gk::init

Handle process startup and initialization

Synopsis

gk::init universal *args*
gk::init faceless ?*value*?
gk::init connectToRegistrar
gk::init acknowledgeToLauncher

Description

The gk::init command is used internally to manage the startup and initialization of GroupKit processes.

The "gk::init universal" command is normally the first thing run by a GroupKit process. It parses the command line parameters and the preferences file. It establishes the gk::localInfo environment, and stores a variety of information there, including the local process' host, port, registries and so on.

The "gk::init faceless" command determines whether the process will run as a background daemon, or will present a user interface. Passing a value of "1" signifies the process will run as a daemon. Calling this routine without any value will return the current value of 1 or 0.

The "gk::init connectToRegistrar" command initiates a connection to the registrar (whose location was determined by gk::init universal, either from the command line, a prompt dialog, or the preferences file). This command also determines the id of the local process, as well as its group.

The "gk::init acknowledgeToLauncher" is needed when the local process has been created at the request of another process, e.g. a session manager used a process creator to create a conference process. This routine connects back to the process launcher to inform it that the process was successfully created, and passes along the process id.

---

gk::shutdown

Handle process termination and cleanup

Synopsis

gk::shutdown exit
gk::shutdown processdied *id*

Description

The gk::shutdown command is used internally to manage the orderly shutdown of GroupKit processes.

The "gk::shutdown exit" command provides a clean way to exit the current process.

The "gk::shutdown processdied" command is typically invoked by portions of GroupKit that maintain socket connections (e.g. the subnets of an environment), when they notice a socket to another process has been closed. This routine generates a "processShutdown" event which can be used to perform cleanup tasks. For example, any nodes in environments that are marked as dependent on the process will be deleted.

---

gk::service

Locate and connect to specific facilities offered by other processes

Synopsis

gk::service provider *providerName* -type *type* -handler *handler* ?-registry *registry*? ?-name *name*?
gk::service subscriber *subscriberName* -type *type* ?-name *name*? ?-registry *registry* ?-host *host*? ?-socket *socket*?

Description

The gk::service command provides an interface to both providing and using services. A service collects a well-defined set of functions that can be hosted on one process and accessed from another. Service requests may be either synchronous or asynchronous.

Providing a Service

The "gk::service provider" command is used to create a new service provider. This creates a new Tcl command named "providerName", which is used to refer to the service provider. Details of this command are below. The mandatory "type" parameter is used to identify the set of requests available; all service providers of the same type should share the same set of requests. The "handler" parameter specifies a Tcl procedure which will respond to requests from subscribers to the service; this is detailed below. The optional "registry" parameter indicates that the service is to be published in the specified registry; otherwise the default registry is used. Finally, the "name" parameter specifies the name under which the service should be published; if omitted, the type is also used as the name.

The Tcl command called "providerName" is used to refer to the service. It has the following two variants:

providerName retract
providerName ?request? ?-async? *request args...*

The "retract" command specifies that the service is no longer available, and should be withdrawn from the registry it is published in. The second form allows the local process to make requests of the provider, without

creating a separate subscriber. Because the request is local, it is inherently handled synchronously; however the -async parameter remains for syntax compatibility with the service subscriber command.

The procedure that handles requests for the service provider must accept certain parameters. The first parameter is always the name of the service provider (i.e. providerName). The second parameter is the name of the request to be made. The valid requests are dependent on the type of the provider, however all providers must handle a request named "requests" by returning the list of possible requests. Finally, other parameters (e.g. args) may be required depending on the particular request.

Subscribing to a Service

gk::service subscriber *subscriberName* -type *type* ?-name *name*? ?-registry *registry* ?-host *host*? ?-socket *socket*? The "gk::service subscriber" command provides a mechanism to locate and then subscribe to a service offered by a provider. This creates a new Tcl command called "subscriberName" which is used to access the service; this is detailed shortly. The mandatory "type" parameter is used to specify the type of service to connect to. The remaining options are used to help narrow down which of possibly many services of the given type to connect to. The "name" parameter specifies that the service should have the given name. The "registry" parameter specifies what registry to search for the service in; normally this is the default registry. The "host" parameter specifies that the service provider should reside on the given host. Finally, the "socket" parameter specifies that the service provider exists on the other end of the given socket. If no matching service provider is found, an error is returned.

The Tcl command called "subscriberName" is used to refer to the service. It has the following two variants:

> *subscriberName* unsubscribe
> *subscriberName* ?request? ?-async? *request args...*

The "unsubscribe" command indicates the caller no longer needs the service; the connection is removed and the Tcl command "subscriberName" is destroyed. The second form makes requests of the provider. By default these are synchronous requests, but this can be overridden with the "-async" flag. The "request" parameter specifies the name of the requests, with any additional parameters needed by the request following.

---

gk::awareness

Support for maintaining workspace awareness information

Synopsis

gk::awareness init
gk::awareness model
gk::awareness registerScrollableWidget *widget ?document?*
gk::awareness documentForWidget *widget*

Description

The gk::awareness command is a preliminary framework used to maintain workspace awareness information on conference participants.

The "gk::awareness init" command initializes these facilities, while the "gk::awareness model" command returns the name of the environment used to keep track of awareness information.

The "gk::awareness registerScrollableWidget" associates a scrollable Tk widget (such as a text or canvas) with an abstract document. If omitted, it uses a "default" document. All widgets that refer to the same abstract document will share awareness information, even if the widgets themselves have different names. This command sets up widget bindings (the -xscrollcommand and -yscrollcommand configuration options) so that when the widget scrolls this is intercepted by the awareness model and shared with others. The "gk::awareness documentForWidget" command is used to determine the name of the abstract document referred to by a widget.

Currently, the position of each user is stored within the awareness model environment under the key *usernum*.view.*document*.viewExtent.

---

gk::base64

Base64 encoding and decoding of binary files

Synopsis

gk::base64 encode *filename*
gk::base64 decode *string filename*

Description

The gk::base64 command provides a mechanism to encode binary files into an ASCII representation, and then decode them. This can be necessary because GroupKit's communications facilities do not yet deal with binary data.

To encode a file, use the "gk::base64 encode" command, which will return a string containing the base64 representation. To decode this representation back into a file, use the "gk::base64 decode" command.

GroupKit Reference Manual. Last updated May 7, 1998 by Mark Roseman.

idgenerator service

Assign unique identifiers

Description

The "idgenerator" service provides a central facility for generating new unique identifiers. It takes a single request named "nextid" which takes no parameters, and returns to the caller a unique identifier.

GroupKit Reference Manual. Last updated April 24, 1998 by Mark Roseman.

environmenthost service

Host centralized environments

Description

The "environmenthost" service provides a central facility for hosting centralized environments. It takes a single request named "create" with a single parameter specifying the name of the environment to create.

GroupKit Reference Manual. Last updated April 24, 1998 by Mark Roseman.

launcher service

Create new conference processes

Description

The "launcher" service provides a facility to create new conference processes on the service provider's host. The primary request is named "create" which is used to create a new process. Its first parameter is the type of conference to create, and additional parameters may also be specified which are passed to the conference. This command will return the process id of the conference. An internal request named "completed" is used by the newly created conference process to confirm its creation with the service.

Note that GroupKit currently provides two implementation of this service; one launches completely new processes, while another creates a separate interpreter running in the same process to hold the new conference.

GroupKit Reference Manual. Last updated April 24, 1998 by Mark Roseman.