SatherGui

Benedict Gomes, David Bailey, Matthias Ernst and Holger Klawitter

SatherGui is a graphical user interface based on the Tcl/Tk widget set. SatherGui provides a type safe interface to the inherently unsafe access functions of Tk. The SatherGui comes in two flavors. The standard version links directly to Tcl/Tk. The other version uses a client-server model adn is called DoubleGui; the client need not be aware of Tcl/Tk and communicates with the server over a socket. Much of the Tk widget set has been implemented as a set of classes in Sather, which provide similar convenience and much greater safety.

1.0 Introduction

The Sather Gui is a set of classes supporting to write graphical user interfaces in Sather. The Sather Gui comes in two flavors:

- The standard Library; also called single process Gui. Using this library no Sather code will be executed when the Tk toolkit is doing something. Vice versa the user interface will freeze as long as Sather code is being executed.
- The DualGui; also called two or double process Gui. The above restriction does not exist any more.

This document includes text cobbled together from different souces and is a rough introduction to some of the Gui functionality. Here are some pointers to additional information about the Sather Gui and related topics.

http://www.icsi.berkeley.edu/~sather/ Docmentation/Library/LibraryBrowser	A browser though the souce files of the Sather Gui.
http://www.icsi.berkeley.edu/~sather/ Documentation/Gui/TclTkDocs	A documentation of the Tk toolkit. This is alsmost essential to understand the behaviour of the Sather Gui.
Doc/Installation.html	A guide how to install the Sather compiler and the Gui.

Warning: The Gui does not implement the full interface of the Tk/Tcl toolkit, but it provides a good framework, which should be easily completable.

2.0 Goals

- Simple programming model; application, not user-interface driven.
- Easy to port to different platforms.
- Easy to implement... :-)

- Minimal interference with standard Sather code.
- Relatively type-safe. The application programmer must not have to deal with strings and the run-time errors that would result from doing so. To quote Alex Cozzi: "Keep the bad Tcl code away from the good Sather code"
- Add-on complexity: Functions have "naive" and "full-fledged" versions.
- No Tk/Tcl modifications. We don't want to have to track distributions.

3.0 User's Guide

This section will focus on just the application end and is valid for both the standard and dual process versions. The Gui works quite similar like Tcl/Tk toolkit and you should be able to use the standard Tk man pages to determine the meaning of various options. However, the widget creation protocols are fairly different.

3.1 Compiling with the Gui

You can get access to the Gui classes by using the -gui option when you compile. For instance:

```
cs -main PIZZA_MAIN -gui -o demo
```

For startes there is a bunch of demonstration programs in Library/System/Gui/Demos.

File Name	Main Class	What it does
pizza.sa	PIZZA_MAIN	Demo with bells and whistles demonstration an ordering tool for pizzas.
demo1.sa	DEMO1	Demo for tags bound to graphical objects.
demo2.sa	DEMO2	More complex set of windows. (looks very weird).
demo_listbox.sa	DEMO_LISTBOX	Demonstrates the usage of listboxes.
demo_menu.sa	DEMO_MENU	A small menu.
demo_text.sa	DEMO_TEXT	Demo of insertion and deletion of text
demo_raster.sa	DEMO_RASTER	Cache simulation by David S.

3.2 Starting up

The Gui provides it's own main routine. In normal cases the programmer just supplies an initialization routine called 'init', which sets up the windows and tells which event results in a call to which routine.

The standard way to use the Gui is to include the handy class GUI_UTIL which provides a couple of simple functions to start up the gui and to enter into the main loop. A standard gui program will look like the following:

```
class MAIN is
    include GUI_UTIL;

    init is
        ... create widgets, bind events to routines, etc.
    end; -- init

end; -- class MAIN
```

3.3 Creating widgets

Creation examples

```
c ::= #TK_CANVAS; -- Create a simple canvas
c ::= #TK_CANVAS(root_window,TK_CANVAS_CFG::std);
-- when you include GUI_UTIL, root_window is predefined.
c ::= #TK_CANVAS(root_window,
                 CANVAS_CFG::std.height(50.0).width(50.0),
                 TK_PACK::left);
```

The above code fragment creates a canvas that is the left part of the root window and does not grow from its initial size of 50 by 50. All widgets (except for the root window widget) support several creation protocols.

Arguments to create	Meaning
None	Plain create with the 'root_window' as parent
parent:$TK_WIDGET, config:CFG	Specify parent (containing) widget and configuration
parent: $TK_WIDGET, pack: TK_PACK	Specify parent and packing information
parent: $TK_WIDGET,config: CFG, pack: TK_PACK	Specify both configuration and packing.

Please see also the file demo files in Library/System/Gui/Demos directory.

3.3.1 Configuration

The options to a widget are all bundled into a config object, which specifies features such as the width, height etc. of the widget. The configuration class for any given widget class usually has the name of the widget followed by an _CFG. You can create a "standard" configuration object with default values, by calling the "std" function on the config class. Starting from this standard configuration, you can then modify various parameters such as the width and height.

The CFG classes are implemented using a large partial class TK_WIDGET_CFG_INCL, which defines all the standard Tk configuration options, but keeps them private. Any particular widget configuration class, such as TK_CANVAS_CFG includes this partial class and makes the relevant options public by renaming them. Thus, you can tell what options are available in a particular configuration class by looking at the list of routines that have been renamed to be public.

Each widget class has a corresponding configuration class which can be used to specify various configuration options.

Widget Class	Configuration Class	Description
TK_FRAME	TK_FRAME_CFG	Frame to hold a collection of sub-widgets
TK_TOPLEVEL	TK_TOPLEVEL_CFG	A special frame that is a top level window
TK_BUTTON	TK_BUTTON_CFG	Simple buttons
TK_LABEL	TK_LABEL_CFG	Labelling widget (similar to an inactive button)
TK_TEXT	TK_TEXT_CFG	Text widget with optinoal scrollbars
TK_CANVAS	TK_CANVAS_CFG	Canvas widget with optional scrollbars
TK_LISTBOX	TK_LISTBOX_CFG	Listbox (standard form has vertical scrollbar)
TK_MENU	TK_MENU_CFG	Menus (can be cascaded). No packing needed!
TK_MENUBUTTON	TK_MENUBUTTON_CFG	Buttons associated with menus
TK_ENTRY	TK_ENTRY_CFG	Widget for entering text.
TK_RADIO	TK_RADIO_CFG	Radio buttons
TK_CHECK	TK_CHECK_CFG	Check buttons

3.3.2 Packing

How the widget is located within its parent frame is specified by the packing widget with the TK_PACK class. I have attempted to simplify the packing configuration options into the following sets of possibilities. First of all, each widget attaches to a particular side (left right top or bottom) of its parent and specifies its own desired width and height (in its configuration options). If the widget should grow with the parent then we must specify in which directions it should grow. The attached side determines the "strong" dimension along which the widget is guaranteed to grow, while the other dimension is a "weaker" dimension. The widget may get left over space in the weaker dimension..

Packer configuration	Description
left	Pack object to the left, don't let it grow to occupy space
right,top,bot	Pack object to the right, top or bottom, similar to above
left_grow_vert	Attach widget to the left and let it grow strongly in the vertical dimension
right_grow_vert	Attach to the right and let it grow strongly in the vertical dimension
top_grow_horiz	Attach to the top and let the widget grow strongly in the horizontal dimension
bot_grow_horiz	Similar to the above
left_n_grow_horiz	Attach to the left and let the widget grow (weakly) in the horizontal dimension
top_n_grow_vert	Likewise with bot_n_grow_vert and right_n_grow_horiz
left_grow_vert_n_horiz	Attach to the left, grow strongly vertically and weakly horizontally.
top_grow_horiz_n_vert	Likewise with bot_grow_horiz_n_vert and top_grow_vert_n_horiz

If the widget does not behave as you wish, I strongly recommend that you read the packing algorithm, either as described in pack.sa or in the tk man page on the packer.

3.4 Events (TK_EVENT,TK_MOUSE_EVENT,TK_KEY_EVENT)

X events are used to trigger various callbacks or other behaviour by "binding" them to a Sather closure (also known as bound routine). Events are specified using the TK_EVENT class. Events in Tcl/Tk are strongly compositonal and consist of several components which can be combined in arbitrary ways. The components of the event are specified using objects of different types - the types enforce the constraints in the manner in which the events may be combined.

Events are of three types - generic events, mouse clicks and button events. Each event has various forms and each form can have an optional "detail" field that more precisely specifies the kind of event, such as which mouse button should be pressed or which keyboard character should be typed

Class	Creation Routines	Optional Detail	Description
TK_EVENT	Motion	None	Mouse motion
	Widget_enter, Leave, Destroy,Visibility,Expose	None	
TK_BUTTON_EVENT	Press or Release	plus(button)	Mouse clicks
	B1, B2, B3	None	Shorthand for mouse clicks
TK_KEY_EVENT	Press or Release	plus(char)	Keyboard key
	Return, Backspace,Tab Linefeed,Up	None	Short forms for special keys

Each event can have one of a set of event modifiers - conditions that must be true for the event to trigger. The event modifiers are

- *Control, Meta, Shift, Alt, Caps_lock* which indicate that that keyboard must be held down during the event.
- *Mouse_1, Mouse_2, Mouse_3* which indicate that the corresponding mouse button must be depressed for the event to trigger.
- *Double, Triple* which means that the event must occur twice or three times quickly for the event to trigger

Example	Triggers when:
TK_EVENT::Widget_enter	The relevant widget is entered
TK_BUTTON_EVENT::Press+1	Mouse button 1 is pressed
TK_BUTTON_EVENT::B1	Short hand - mouse click with button 1 (use this version usually)
TK_BUTTON_EVENT::B1.Double	Double click with mouse button 1
TK_KEY_EVENT::Press+'c'	Type the keyboard character 'c'
TK_KEY_EVENT::Press.Alt+'c'	Type Alt and the character 'c'
TK_KEY_EVENT::Return	Type return
TK_KEY_EVENT::Return.Alt	Type return while holding down the ALT key

Please the Tk man page on "bind" for more details.

The GUI_UTIL provides some basic events like quit_routine.

3.5 Canvas widget (TK_CANVAS)

The Tk canvas widget is an extremely verstaile drawing tool. The Sather TK_CANVAS consists of an outer frame with optional scroll bars, along with a Tk canvas widget. Since we don't (yet) provide direct access to Tk scrollbars, this arrangement is more convenient.

3.5.1 Creation

```
c ::= #TK_CANVAS;
c.draw_line(0.0,0.0,10.0,10.0);
c.draw_text("test",3.0,3.0);
```

This will create a line and draw text at the specified location. Locations are pixel locations, with 0,0 being the upper left corner.

3.5.2 Configuring and tagging items

Lines and other canvas items may be configured and tagged. Canvas tags are of the class TK_CTAG

```
my_line_tag ::= #TK_CTAG; -- Canvas tag
config ::= TK_LINE_CFG::std.fill_color("blue");
c.draw_line(0.0,0.0,10.0,10.0,|my_line_tag|,config);
```

Canvas items, like standard widgets, each have a configuration class which may be used to specify various configuration options which can be later changed:

Item type	CFGuration class	Description
Lines	TK_LINE_CFG	Lines, including lines with arrows and splines
Rectangles	TK_RECT_CFG	Rectangles
Ovals	TK_OVAL_CFG	Circles and other ovals
Embedded windows	TK_CWIND_CFG	Arbitrary embedded windows
Text	TK_CTEXT_CFG	Embedded text in the canvas

Canvas items have the following routines - a plain item creation routine, item creation along with configuration and tag specification and item reconfiguration. For example:

```
draw_text(text: STR, atx,aty: FLT);
draw_text(text:STR,atx,aty: FLT,tags: ARRAY{TK_CTAG},config: TK_CTEXT_CFG)
text_configure(tag: TK_CTAG, config: TK_CTEXT_CFG);
```

Just as with standard widget configuration classes, item configuration classes provide a "std" creation routine which returns a useful set of defaults.

3.5.3 Setting up a canvas callback

The goal is to call function "do_something" when you click on the line.You must first associate a tag with the item. New tags are created by a creating a TK_CTAG. For those of you familiar with Tcl/Tk, each new TK_CTAG corresponds to a unique string.

```
my_line_tag ::= #TK_CTAG; -- Canvas tag
config ::= TK_LINE_CFG::std.fill_color("blue");
c.draw_line(0.0,0.0,10.0,10.0,|my_line_tag|,config);
c.item_bind(TK_BUTTON_EVENT::B1,-- Event is a button 1 press
   my_line_tag,            -- For items tagged my_line_tag
   bind(do_something(_))) -- Trigger this rotuine
```

And here's the actual callback routine:

```
do_something(a: TK_CANVAS_CB) is
    -- Canvas callbacks get an argument of type CANVAS_CB
```

```
        -- Please look at the class CANVAS_CB to see the information
        -- provided to the callback routine.
       #OUT+"Reached here!";
   end;
```

3.6 Text widgets (TK_TEXT)

Text widgets are slightly different from the standard Tk text widgets - they contain an outer frame with (optional) scrollbars in addition to the text widget itself. It is possible to access the Tk text widget itself, if that is necessary.

3.6.1 Creation

```
t ::= #TK_TEXT;
t+"test1";
```

The plus routine basically appends text to the end of the text widget. Scroll bars may also be specified or avoided. The following line creates a text window with no vertical scroll bar:

```
t ::= #TK_TEXT(root_window, TK_TEXT_CFG::std.vscroll(false));
```

3.6.2 Specifying text at a particular location

You can specify text at a particular location in a text widget by using TK_INDEX to indicate text locations

```
t.insert_at(TK_INDEX::at(10,1),"My insertion at 10,1");
t.delete(TK_INDEX::at(11,2),TK_INDEX::at(11,10));
```

TK_INDEX has the following protocol:

Location specification	Meaning
at(line,char: INT)	Location at line "line" (starting from 1) and character "char" (starting from 0)
end_pos	Last location in the text
first(tag: TK_TEXT_TAG)	Location of first character associated with a given tag
last(tag:TK_ TEXT_TAG)	Location of last character associated with a given tag
mark	Current location of the insert cursor.

Text indices may also have modifiers such as forward_chars(i: INT), backward_chars, forward_lines,backward_lines,line_start, line_end, word_start and word_end.

For example:

```
TK_INDEX::at(10,10).word_end;   -- End of word located at line 10, char 10
TK_INDEX::mark.line_start; -- Beginning of the line with the current mark
```

3.6.3 Text tags

Tags (of type TK_TEXT_TAG) may be used to mark certain portions of text which may then be manipulated in various ways.

3.7 Window manager interface (TK_WINDOW_MGR)

Some of the window functionality is not part of any Tk widget. These calls are being handeled by the TK_WINDOW_MGR class. Currently one can only set the title of a window with it.

```
TK_WINDOW_MGR::title(tplevel_widget,"Title");
```

3.8 Tk Variables (TK_VAR)

Tk supports variables which can be connected to particular widgets. The advantage of these agains normal Sather variables is that that Tk notices changes to them and updates the involved widget automatically. The TK_RADIO and TK_CHECK classes tahe heavy isage of them.

3.9 Check buttons widget (TK_CHECK)

A check button is a button which can be selected or deselected. The user can link a TK_VAR to the check button so that the value of the button get updated when the status of the button changes.

Acces routines	Description
TK_CHECK_CFG::variable(v:TK_VAR)	Associate the variable v with the check button.
TK_CHECK_CFG::on_value(s:STR)	Associate a value when selected.
TK_CHECK_CFG::off_value(s:STR)	Assocuate a value when deselected.
TK_CHECK::select	Select the button
TK_CHECK::deselect	Unselect the button
TK_CHECK::toggle	Switch between selected and deselected.
TK_CHECK::command(r:ROUT)	Associate an action for when the button is being toggled.

3.10 Radio button widget (TK_RADIO)

In opposite to the check buttons multiple radio buttons can share the same TK_VAR. Only one of the radio buttons can be selected at a time. The associated variable has the value associated to this radio button.

Routine	Description
TK_RADIO_CFG::variable(v:TK_VAR)	Associate a variable with the radio button
TK_RADIO_CFG::val(v:STR)	Associate a value for the variable
TK_RADIO::select	Select the radio button. Deselect all other readio buttons associated to the same variable.
TK_RADIO::deselect	Delselect the radtio button.
TK_RADIO::command(r:ROUT)	Associate an action when the radio button is selected.

3.11 Debugging the interface

If you really want to know what happenes between the Tk and Sather side of the program you can force the Sather Gui to print the messages running between them. This can be done by setting the Tk variables in the Tcl files in SATHER_HOME/System/Platform/gui/TclCode. One can set the following Tk variables:

Flag name	In file	Action
server_debug	gui_setup.tcl	Turn on server end debugging (early)
server_tcldebug	gui_setup.tcl	Print out tcl that reaches server end

Flag name	In file	Action
app_debug	gui_post_socet_int.tcl	Turn on application end debug (starts a bit late)
app_tcldebug	gui_post_socet_int.tcl	Print out only tcl that the application sends

4.0 Some Implementation Details

4.1 Directories

The following are the directories that consistute the Gui.

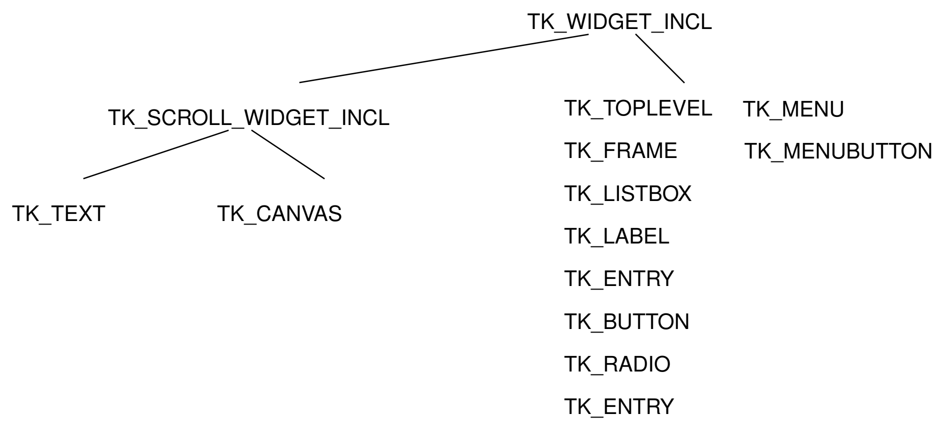
Directory	Purpose/Contents
Library/System/Gui	The Sather code for the Gui.
Library/System/Gui/Demos	The demos described above.
System/Platforms/gui	Neccesary changes to the generation process of Sather programs using the Gui.
System/Platforms/tcltk	The tcltk Platform code - tcl/tk initialization code
Library/System/TclTk	Sather source for the interface to Tcl/Tk
Library/System/DualGui	Sather classes differing from the ones for the Single Gui.
System/Platforms/dualgui	Neccesary changes to the generation process of Sather programs using the DualGui. The Sather code of the server is also here.

4.2 Differences from Tcl/Tk

- Text and canvas widgets are not really tk text and canvas widgets - they have an outer “frame” which holds the real widget as well as optional scroll bars.
- We don’t rely on string names to identify objects. Even in places where the user can provide a string name, this name is used for debugging purposes only - internally, it is made unique. The sather object identity should be used. TAG objects, for instance, are unique. We make them unique inSather by appending a unique integer. If you want to use the same tag in a couple of places, use the same sather tag object.

4.3 Widgets

All widgets, directly or indirectly, include the TK_WIDGET_INCL partial class. This partial class provides the basic creation routines and has stubs for the name of the widget being created. The actual widget class must fill in these names. Scrollbar widgets have not yet been implemented directly. However, scroll bars are provided as a standard option on the text and canvas widgets. These scrollable widgets include the partial class TK_SCROLL_WIDGET_INCL .



4.3.1 TK_WIDGET_INCL

An include class used by widgets. Various shortforms for evaluating using a bunch of arguments (concatenating arguments is prone to the error of ommitting a space between the arguments). It also provides the basic creation structure for widgets. The create routine is specified in the widget itself. It is recommended that one looks at a simple widget like TK_LABEL or TK_BUTTON to see how the included code is used.

4.3.2 TK_SCROLL_WIDGET_INCL

An include class used for widgets which might have scrollbars. This creates an outer frame which can hold the widget proper as well as the scrollbars. Take a look at Widgets/text.sa TK_TEXT.

4.3.3 TK_WIDGET_CFG_INCL

Each widget has a corresponding CFG class which is used to specify and change configuration options. Most configuration options are actually specified in TK_WIDGET_CFG_INCL and are kept private. On inclusion, the configuration options that are relevant to a particular widget are made accessible by making them public. All configuration options have a version that returns self and one that does not, to permit convenient dotting.

4.3.4 TK_ARG_UTIL

"Macros" for concatenating strings, some of which might be void.

4.3.5 Global Table: TK_WIDGET_MAP

Mapping from widget names to widget objects used by GUI_APP_END.

4.3.6 Per-widget Table:TK_WIDGET_CALLBACKS

Mapping from action_id s to bound routines and also to the name of the action (for debugging/error reporting). *This class should go away, as during the development of the Gui the event system had undergome improvements which make this class unneccessary.*

4.4 Events

Events are being processed in the following way:

1. (In $TK_EVENT::create) Each event class has a different callback string called "cb_str". This string encodes all valid event information which can be received when the event occurs. Internally the callback string contains all possible inforation options. The irrelevant options are masked out with an at-sign (@).
2. (In $TK_WIDGET::bind_event) The widget to which the event is being bound in "bind_event" stores all events in a list (independantly from the Tk event scheduling system). The index of the event within the listdefines the event class. (Hence events cannot be removed right now.)
3. When an event occurs, Tk calls "act_on" withing the widget and passes the event class along with the list of evaluated callback string. The Tk side evaluateds only arguments which have not been masked out.
4. (In $TK_WIDGET::act_on) The "act_on" routine creates a TK_EVENT_INFO and fills the arguments into it. They are not yet being evaluated, because the user probably will not use all of themand the computation time wouldbe mostly wasted.
5. Then the associated routine from "bind_event" is being called. The routine receives the event information for further processing.

4.5 The Tcl/Tk Code

The interface to Tcl/Tk is in via the tcltk platform. See System/Platforms/tcltk. The init subdirectory contains a copy of the system initialization files for Tcl/Tk and are copies of the ones in the distribution. These are copied here to avoid having to specify yet another system dependancy. The Platform.module file specifies particular locations of the Tcl/Tk libraries and header files.

The Tcl code is in the System/Platform/gui directory and consists of initialization files for the gui.

File name	What it does
gui_setup.tcl	Sets up the basic callback timer from Tk. Adds some procedures to make callbacks simpler
gui_post_socket_int.tcl	Sets some more server end debug flags. In the case of the dual process Gui, these debug flags must be set after the connection is established. This is why they are separated from gui_setup.tcl

5.0 The Dual Process Gui (DualGui)

The two process version of the Gui makes it possible for application programs to support rapid updating of on-screen graphics without having to litter the application code with calls to an event-processing routine (though you still need calls to a polling routine to handle callbacks, these polls can be much less frequent since many simple events are handled independantly). This is accomplished by having a separate process for maintaining the graphics. Communication between the application program process andthe GUI "server" process is done via sockets. As a result, you have to be aware of a few choices which must be made when initializing the GUI.

The essential benefit of the dual process version is that you can retain control in your Sather program i.e. you need not call gui_loop. It is sufficient if you call gui_poll frequently enough to permit reasonable updates.

5.1 Separate Machine Method

It is possible to run the GUI server process on a separate machine from the application program process. This can be useful if the application program runs on specialized hardware which does not support X Windows. It is also useful when running an application program on a remote machine over a slow link, since the bandwith used by the Sather GUI can be orders of magnitude lower than the corresponding X bandwidth, since many mouse and other events are handled at the server end.

To start the GUI this way, your application program should call GUI_UTIL::startup_gui(false,true), and you should arrange that after that call is made, the program $SATHER_HOME/System/Platforms/dualgui/dual_gui_server <application-host-name> is run on the local machine. (Actually, you can run the server up to several seconds before the applicationprogram and all will still be well.) Internet-domain sockets are used in this case.

5.2 Debugging Method

For debugging, you may wish to separate GUI server error messages from your application program error messages. To do this, you may follow the instructions in the above paragraph, using two xterms logged into the same machine, and using "localhost" as the host name.

If it is important to retain the communication speed of the Unix-domain sockets, start the GUI with GUI_UTIL::startup(false,false) and omit the hostname when invoking gui_server_end.

5.3 Forcing Internet Sockets

Lastly, should you wish to use the usual, server-forking, startup procedure but want to use Internet-domain sockets for some reason, you can simply start the GUI with GUI_UTIL::startup(true,true).

5.4 Notes

1. When using Unix-domain sockets, there is an extremely small but non-zero chance that an ill-timed interrupt of your application program could leave some "junk" in the filesystem preventing future runs of any Sather GUI program on the machine. In this case, delete the files /tmp/GUI_SYNCH and /tmp/GUI_ASYNCH, and report the problem to dbailey@icsi.berkeley.edu.
2. If you suspect you've got socket-related problems, refer to the files socket.sa and socket_support.c in $SATHER_HOME/Library/System/Socket, or contact dbailey@icsi.berkeley.edu.

6.0 The SOCKET Layer

The basic communication between the user interface client and the actual display will usually take place over a socket based interface.

6.1 Reasons for using sockets

This socket interface might seem an arcane structure, but there were several reasons for it. This setup was designed by David Bailey and these are mostly his motivations:

- The programming model is cleaner, and application rather than interface driven. The application runs and generates the user interface and sets up callbacks. Occassionally the application has to poll to allow the callbacks to be serviced. The standard approach is to drive the application from the user-interface, which is not appropriate for visualization of a principally computational task. This approach essentially gives you multi (two!) threading.

- This model permits communication between a gui that runs on a server machine and communicates with an application on a different machine, which possibly does not understand X. This was particularly important when we were considering a GUI for use with Sather running on specialized hardware. This could prove quite useful for using X from home over a (relatively) slow modem. David already reports that his thesis interface runs very well (much faster than X) when used from home while running at ICSI. I would be very interested in seeing the performance of a simple editor that uses this philosophy - it should be much faster than standard editors, since all scrolling etc. will be local.
- And this is perhaps a trivial reason, but was a major reason for me! I found that on some occassions gdb would not run on the browser - I could only conclude that it was due to some interaction with tcl, maybe with the two garbage collectors? This socket version seems much safer, since tcl/tk is separate from the application.
- The socket interface is hidden and quite small; if someone knows how to get the same programming model and safe gdb usage without separating out the processes (threads in pSather, for instance), it should be possible to create a drop-in non-socket replacement with just a couple of hundred lines of code.
- Linking time is somewhat reduced, and the executable for the tk end can be distributed.

On the minus side, this design is somewhat more complex and quite a bit less efficient than a single process solution.

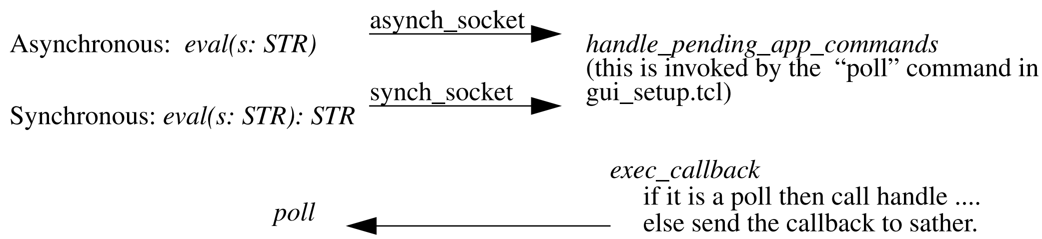
6.2 Socket management

At the sather end, the socket communication is managed by GUI_APP_END, whose main routines are "startup" and "poll". There is a corresponding GUI_SERVER_END in the tk code.The server end consists of an interface to Tcl (taken from Tk/Tcl Kit by Matthias Ernst and Alex Cozzi) and an interface to a pair of sockets (David Bailey). The socket level consists of two sockets

Creation of the application end consists of spawning off the server and then creating the asynch and synch sockets. The server connects to the client sockets and then sources gui_init.tcl and gui_setup.tcl which sources other tcl files and sets the value of various debug flags.

A callback is sent by GUI_SERVER_END::exec_callback over the synch socket to the sather process. There it will wait until the sather program does a "GUI_APP_END::poll" command. The poll command will process the arguments

- The first argument is the widget name (or possibly a special debug flag)
- The second argument is the number of arguments. Then comes action id and the actual arguments
- The appropriate sather widget is found using the callback_map
- act_on is called on the target widget with the action id and the argument array, which starts with the action id. Widgets will usually convert this structure into something such as TK_CANVAS_CB in which the arguments have been processed. The interpretation may differ from widget to widget (and possibly from event to event)



- act_on will then locate an appropriate bound routine corresponding to that action id and call that bound routine with the processed callback arguments.
- All widget classes have several eval routines which call the asynchronous interface to Tk.

7.0 Acknowlegements

- Alex Cozzi (TclKit comments and suggestions)
- David Stoutamire (examples and feedback)