Building Tcl-Tk GUIs for HRT-HOOD Systems

Juan Carlos Díaz Martín — Isidro Irala Veloso — José Manuel Rodríguez García

Departamento de Informática, Universidad de Extremadura. Avda. de la Universidad, s/n, 10071, Cáceres, Spain.

34 927 257265 — 34 927 257265 — 34 927 181092

juancarl@unex.es — iirala@unex.es — jmrodri@unex.es

1. ABSTRACT

This work explores Tcl-Tk 8.0 as a building tool for script-based GUIs in Ada95 real-time systems. Tcl-Tk 8.0 is a library that makes graphic programming easier, but it suffers from being non-thread-safe. An application architecture is proposed, the deferred server, which provides transparent use of Tcl-Tk to multithreaded Ada95 applications via TASH, a thick binding that allows Ada95 single-threaded code to use Tcl-Tk. We find that only a minimal extension to Tcl-Tk 8.0 and TASH is required to support it, while a successful prototype has been implemented based on these ideas. Likewise, the early integration of Tcl-Tk graphic user interfaces in HRT-HOOD designs is examined; unfortunately, in this respect, we conclude that this is not feasible. However, a HRT-HOOD conform distributed configuration is outlined in wich the user interface becomes a multithreaded remote service based on the deferred server architecture.

1.1 Keywords

Real time systems, User interfaces, Tcl-Tk, HRT-HOOD.

2. INTRODUCTION

User interface is generally a neglectic topic in real time methodologies. Yet, both in industry and high education, monitoring and supervising the evolution of the system under development through a friendly graphic environment is important. A good user interface can provide a graphic model of the system evolution; it also facilitates its study and familiarization, and it makes its development a more attractive task. This last issue is particularly important in the university environment, where there is generally little time to build a running non-trivial system ([3]); it is even harder to learn and program on a particular windows system. Here, Tcl-Tk scripts seem to be a balanced compromise of power, flexibility and ease of use. In the industrial side area, in a true hard real time system, the pair (System, User Interface) must be analizable in its temporal constraints.

This paper presents the following: Firstly, a method to use Tcl-Tk scripts from concurrent Ada95 applications, and, secondly, an extension of the method to HRT-HOOD systems, by introducing the user interface early in the design stage to make the pair (System, User Interface) temporally analizable as a whole. The three main components are shown in the Fig. 1.

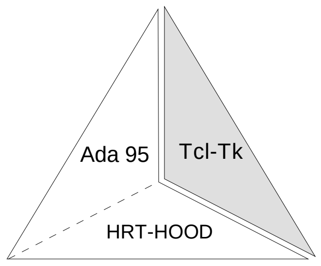

Fig. 1 The main problem components.

As HRT-HOOD automatically translates to Ada 95, the frontier between both is represented by a slim broken line. Unfortunately, Tcl-Tk doesn't directly fit. The Tcl-Tk fitting problem has two sides, namely, the concurrent use of Tcl-Tk, and the design restrictictions of HRT-HOOD systems.

The rest of the paper is organized as follows: Section 2 summarizes Tcl-Tk and TASH, and section 3 presents the current work on Tcl-Tk to make it thread-safe and its impact on our work. The Ada95/Tcl-Tk side is addressed in sections 4 and 5, and the one dealing with HRT-HOOD/Tcl-Tk, in section 6. Section 7 describes our current work for making a distributed GUI for HRT-HOOD systems. Finally, conclusions are presented in section 8.

This work was in part supported with U.E. FEDER-II founds (Proyect S742-F96) and Junta de Extremadura.

3. ON TCL-TK AND TASH

Tcl is a general purpose interpreted command language that admits specialized extensions, being Tk the most known and used. Tk enriches Tcl with a set of commands for using the underlying graphic platform. This augmented Tcl is known as Tcl-Tk. Tk's greatest virtue is probably its ease of use; two or ten lines are enough to make simple applications run.

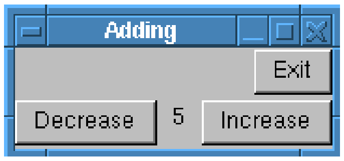

Fig. 2 User interface built by `Adding.tcl`

Code 1 is a Tcl-Tk script that builds the user interface of Fig. 2.

```
1   #! /usr/bin/wish
2   wm title . Adding
3   frame  .menu -width 50 -height 10
4   button .menu.exit -text Exit \
5     -command exit
6   frame  .frame -width 50 -height 40
7   button .frame.incr -text Increase \
8     -command increase
9   button .frame.decr -text Decrease \
10    -command decrease
11  label .frame.lab -text { }
12  pack .menu -fill x -expand true \
13    -side top
14  pack .menu.exit -side right
15  pack .frame -fill both -expand true \
16    -side top
17  pack .frame.incr -side right
18  pack .frame.decr -side left
19  pack .frame.lab
20
21  set count 0
22  #====================================
23  #           increase --
24  #====================================
25  #Show the increased value of count
26  proc increase { } {
27    global count
28    incr count
29    .frame.lab configure -text $count
30  }
```

Code 1 `Adding.tcl`: A Tcl-Tk script

```
31  #====================================
32  #           decrease --
33  #====================================
34  #Show the decreased value of count
35  proc decrease { } {
36    global count
37    incr count -1
38    .frame.lab configure -text $count
39  }
```

Code 1 `Adding.tcl`: A Tcl-Tk script (Cont.)

The application starts when, at the command line, the Tcl-Tk interpreter is invoked:

```
$ wish adding.tcl
```

`Wish` is the Tcl-Tk interpreter, supplied with the Tcl-Tk distribution. Clicking the "Increase" and "Decrease" buttons changes the counter placed between them. These buttons are created in lines 7-8 and 9-10. Each click triggers the corresponding procedures `increase` (lines 26-30) and `decrease` (lines 35-39). Code 1 allows us to distinguish between two kinds of commands; first, Tcl-Tk *Built-in* commands, such as `button`, and, secondly, user written commands, such as `increase` and `decrease`, known as *Application Specific Commands.*

As the Tcl-Tk interpreter can be embedded in a C program, our approach is to use the Tcl scripting language in concurrent applications. The Tcl `Tcl_Eval` function allows C code to invoke a Tcl-Tk script. For example, to create the Fig. 2 "Increase" button from a C program, we will use:

```
Tcl_Eval(&Tcl_Interp,
  "button .frame.incr -text Increase \
     -command increase");
```

where `Tcl_Interp` is a handle to a Tcl-Tk interpreter.

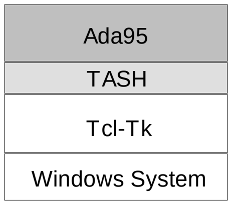

Fig. 3 TASH, a thick binding Ada95/Tcl-Tk

The Tcl-Tk interpreter is written in C. It is easy to call it from a C module by `Tcl_Eval`. However, to invoke Tcl-Tk from Ada 95, is rather cumbersome. TASH (from Tcl-Ada-SHell) is a thick binding Ada95/Tcl-Tk ([8]). As Fig. 3 shows, TASH provides Ada95 applications with the whole Tcl-Tk interface by the packages `Tcl` and its children. Code 2 shows the use of TASH. It is the Ada95 replica to the Tcl script Code 1:

```
with Tcl;          use Tcl;
with Tcl.Tk;       use Tcl.Tk;
with Interfaces.C; use Interfaces.C;
with CArgv;
with Tcl.Ada;

procedure adding is
 package C renames Interfaces.C;
 package CreateCommands is new
   Tcl.Ada.Generic_Command(Integer);
 Number_Label : Label;

 function Decrease_Cmd(
  ClientData : in Integer;
  Interp     : in Tcl_Interp;
  Argc       : in C.Int;
  Argv       : in CArgv.Chars_Ptr_Ptr
 )return C.Int;
 pragma Convention(C, Decrease_Cmd);

 function Decrease_Cmd(
  ClientData : in Integer;
  Interp     : in Tcl_Interp;
  Argc       : in C.Int;
  Argv       : in CArgv.Chars_Ptr_Ptr
  )return C.Int is
 begin
  eval("incr count -1");
  configure(Number_Label, "-text
                           $count");
  return TCL_OK;
 end Decrease_Cmd;

 Interp    : Tcl_Interp;
 cmd       : Tcl_Command;
 Incr_Butt : Button;
 Decr_Butt : Button;
 Menu_Fra  : Frame;
 Base_Fra  : Frame;
 Exit_Butt : Button;

begin  --Adding
 --Create interpreter
 Interp := Tcl_CreateInterp;

 --Initialize Tcl
 if Tcl_Init(Interp) = TCL_ERROR then
  Tcl_DeleteInterp(Interp);
  Tcl_Exit(1);
 end if;

 --Initialize Tk
 if Tcl.Tk.Init(Interp)=TCL_ERROR then
  Tcl_DeleteInterp(Interp);
  Tcl_Exit(1);
 end if;

-- Create Ada95 user extended command
 cmd:=CreateCommands.Tcl_CreateCommand(
  Interp, "decrease",
  Decrease_Cmd'access, 0, NULL);
 Tcl.Tk.Set_Context(Interp);
```

Code 2 `Adding.adb`: Embedding Tcl in Ada95

```
-- Create Ada95 user extended command
 cmd:=CreateCommands.Tcl_CreateCommand(
  Interp, "decrease",
  Decrease_Cmd'access, 0, NULL);
 Tcl.Tk.Set_Context(Interp);

 -- Main window title
 wm("title", ".", "adding");

 -- Create user interface
 Menu_Fra     := Create(".menu", "");
 Exit_Butt    := Create(".menu.exit",
   "-text Exit -command exit");
 Base_Fra     := Create(".frame", "");
 Incr_Butt    := Create(".frame.incr",
   "-text Increase -command increase");
 Decr_Butt    := Create(".frame.decr",
   "-text Decrease -command decrease");
 Number_Label :=Create(".frame.number",
   "-text { } ");

 -- Create Tcl-Tk user extended command
 eval("set count 0");
 eval("proc increase { } { " &
   " global count; " &
   "   incr count; " &
   ".frame.number configure -text
      $count; " &
   "}");

 pack(Menu_Fra,     "-fill x
   -expand true -side top");
 pack(Exit_Butt,    "-side right");
 pack(Base_Fra,     "-fill both
   -expand true -side top");
 pack(Incr_Butt,    "-side right");
 pack(Decr_Butt,    "-side left");
 pack(Number_Label, "");

 -- Tcl-Tk infinite service loop
 MainLoop;
end adding;
```

Code 2 `Adding.adb`: Embedding Tcl in Ada95 (Cont.)

Callbacks are the Tcl mechanism that implements the interaction of the user with the application. In the Tcl-Tk jargon, a *callback* is a Tcl command `Cmd`, either a user extended command or a Tcl-Tk built-in command, that is passed into another procedure and is executed later, perhaps with additional arguments ([7]). We now introduce the term *user callback*. A user callback is defined as the invocation of an Application Specific Command written in the host language, either C or Ada95, by the Tcl-Tk library. Fig. 4 shows commands, languages, callbacks and user callbacks.

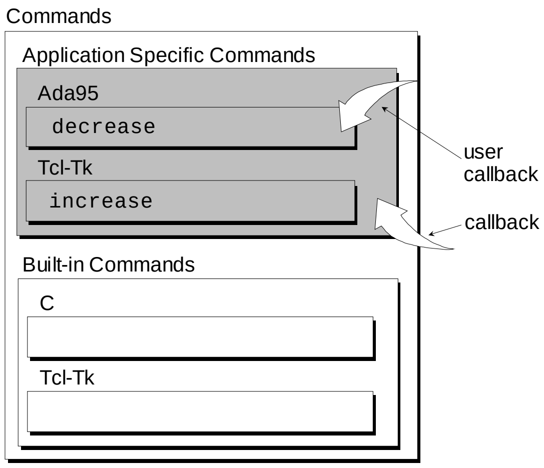

Fig. 4 Languages, commands and callbacks

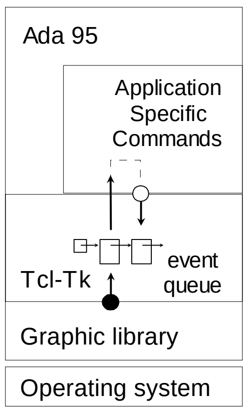

Fig. 5a Classic single-threaded Tcl-Tk architecture

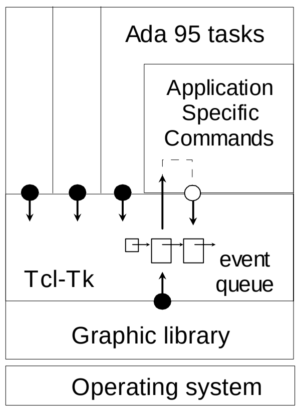

Fig. 5b Multi-threaded deferred server architecture

4. RELATED WORK

The initial goal of this work is to build a mechanism that enables any Ada95 task to transparently execute a Tcl-Tk script using `Tcl_Eval`. At the time of this writing, the Tcl-Tk reentrance is been addressed by its creators. The upcomming Tcl-Tk 8.1 release have reentrant features. It is thread-safe, although in a rather restricted sense: Each thread can have its own interpreters but each interpreter can only be accessed through a single thread. Thus, its true that `Tcl_Eval` can be concurrently invoked from two different threads, but with (and only with) two separate interpreters. That means that every Ada95 task in the system that uses the Tcl-Tk GUI should create its own interpreter before any `Tcl_Eval` call; this complicates programming, but it should not present any major difficulty. In order to share a single intepreter, the Tcl-Tk 8.1 provides new Tcl C-level APIs to send commands and scripts to another thread. Of course, the TASH thick binding must keep up with the new thread-safe 8.1 features in a concurrent Ada95 application. Up to that moment, and even later, we think that the monolithic approach of this work can be useful. It even is mandatory for those restricted to use non-reentrant Tcl-Tk releases.

5. THE DEFERRED SERVER ARCHITECTURE

Service-loop architecture takes control of the thread that executes it, wich prevents Tcl-Tk to be invoked from a multithreaded process. Our effort makes current Tcl-Tk implementations available for concurrent Ada 95 applications in a transparent way.

Firstly, we analize the Tcl implementation. An ordinary Tcl-Tk program consists of a single thread of control that manages an event queue (Fig. 5a), ordered on priority basis. Each event is a data structure comming from two sources, the user code, when `Tcl_Eval` (`Eval` under TASH) is invoked, and the graphic library (X-Windows, for example) on user actions. The event structure has two fields: an event identifier and a pointer to a command. The thread executes the infinite event-loop `Mainloop`, which has the following pseudocode:

```
while(1)
  Do_Event();
```

`Do_Event` access to the event queue shown in Fig. 5 and serves the next event, perhaps by making a user callback. In this case, the invoked Ada95 Application Specific Command may call a Tcl-Tk Built-in command, what causes a new event to be inserted in the queue. When the queue is emptied, the thread calls the graphic library, where the whole application gets blocked waiting for user action. Under the X-Windows system, for example, the client library gets blocked on the TCP connection to the server. On return, data are converted into events and inserted in the queue.

In the classic Tcl-Tk architecture shown in Fig. 5a, the only existing thread takes control, calling the Ada95 Application Specific Commands just on user actions. Our purpose is to do the opposite: Multiple user threads call the Tcl-Tk facilities when needed; i.e., at any time, the `Tcl_Eval` function may be invoked by any thread in order to the interpreter to execute a command (Fig. 5b). Of course, this goal impose the elimination of the infinite blocking Tcl-Tk service loop. The drawback is that each call to the interpreter has the efect of inserting a new event in the queue and, without the service loop, nobody serves the events. The queue grows endlessly.

Our approach is to dedicate an additional *periodic* thread for the event queue testing and serving. We aim not to change the Tcl-Tk event queue model, but, rather, to modify the way of using it. We have worked on the Tcl-Tk 8.0 version of the library. Two routines have been added to it: `Test_Events` and `Do_Events`. Furthermore, `Test_Events` and `Do_Events` have been added to the original TASH interface to be invoked from Ada95.

`Test_Events` is a fast function that inspects both the event queue and the X-Windows server connection. If there are data in the connection or events in the queue, `Do_Events` is called upon. `Do_Events` reads all the data in the connection, transforms them in Tcl-Tk events, inserts them in the queue and, finally, it serves them. Thus, `Test_Events` and `Do_Events` avoid the blocking service loop provided by the Tcl-Tk. The user code does not use them. Under the new architecture, a new thread tests both the event queue and the server connection on periodic basis. This thread is a dedicated periodic task, called `TASH_Handler`, which has the following implementation:

```
package TASH_Handler is
  pragma Elaborate_Body;
end TASH_Handler;

with TASH_Controller;
package body TASH_Handler is
  task TASH_Handler_Th is
    pragma Priority(T_H_PRIORITY);
  end TASH_Handler_Th;

  task body TASH_Handler_Th is
    T: Time;
    Period: Time_Span := T_H_PERIOD;
  begin
    T := Clock;
    loop
      if Test_Events then
        Do_Events;
      end if;
      T := T + Period;
      delay until (T);
    end loop;
  end TASH_Handler_Th;
end TASH_Handler;
```

The `delay until` sentence gives the periodic character to the task and allows the scheduling of other activities. The period of the task depends on the responsiveness desired from the user interface. The more responsive the application, the shorter the period. In any case, keeping `Test_Events` fast, the periodic thread minimally interferes a with the rest of the system. The service loop architecture has the advantage of serving the events immediately. Under the proposed design, however, the events wait for `TASH_Handler` to activate. The application tasks use `TASH`, being unaware of `TASH_Handler`. In addition, the Ada95 application tasks view the same interface for the Tcl-Tk script that the classic single-threaded one addressed by TASH. Because of its nature, from here on, we shall call this design as the *deferred server* (Fig.6).

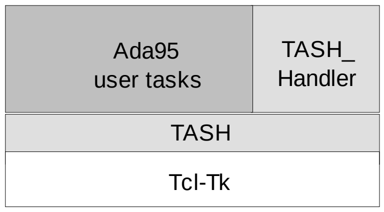

Fig. 6 The deferred server (I)

6. THE REENTRANCE PROBLEM

The periodic thread of Fig. 6 solves the service loop problem addressed in section 5. Yet, the reentrance problem, persists. Tcl-Tk 8.0 is the typical non-thread-safe third party library used by threaded applications. Concurrent invocations of the Tcl-Tk interface cause the **reentrance** in the Tcl-Tk library. The general strategy to continue using Tcl-Tk is to consider the library as a big monolithic monitor.

6.1 The first design attempt

A concurrent application may use the Pthreads interface to support the concurrency. In such a case, the monolithic monitor is implemented as a wrapper at every library access. For example, a multithreaded C program could use Tcl-Tk as follows:

```
pthread_mutex_lock(TclTk_Mutex);
Tcl_Eval(&Interp, "My_Command");
pthread_mutex_unlock(TclTk_Mutex);
```

Luckily, Ada95 applications using TASH can solve the reentrance problem much more cleanly (Fig. 7).

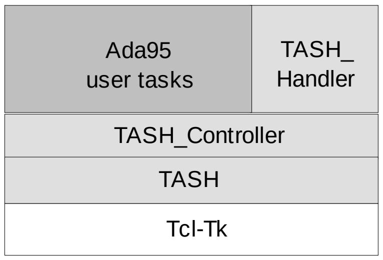

Fig. 7 The deferred server (II)

The key is to introduce a protected object, `TASH_Controller`, that encapsulates the whole TASH interface.

```
with Tcl;
with Tcl.Tk;
package TASH_Controller is
  protected Agent is
    procedure Tcl_Init(...);
    procedure Tcl_CreateCommand(...);
    procedure Tcl_Eval(...);
    procedure Tk_Init(...);
    ...
    procedure Do_Events(...);
    procedure Test_Events(...);
  private
    ...
  end Agent;
  procedure Tcl_Init(...)
    renames Agent.Tcl_Init;
  procedure Tcl_CreateCommand(...);
    renames Agent.Tcl_CreateCommand;
  procedure Tcl_Eval(...);
    renames Agent.Tcl_Eval;
  procedure Tk_Init(...)
    renames Agent.Tk_Init;
  ...
  procedure Do_Events(...);
    renames Agent.Do_Events;
  procedure Test_Events(...);
    renames Agent.Test_Events;
end TASH_Controller;

package body TASH_Controller is
  protected body Agent is
    procedure Tcl_Init(...) is
    begin
      Tcl.Tcl_Init(...);
    end Tcl_Init;
    ...
  end Agent;
end TASH_Controller;
```

Unfortunately, this attempt of solving the reentrancy problem is faulty. Under a Tcl-Tk interface, the user interacts with the system through user callbacks. The rest of this section shows that user callbacks lead to deadlock.

Fig. 8 shows a very simple embedded system, a heater, whose control goal is to keep the water temperature as close as possible to the *r(t)* desired reference.

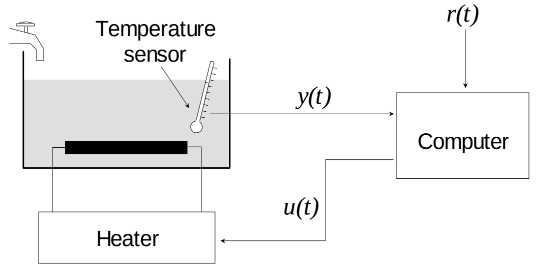

Fig. 8 The heater control system

A Tcl-Tk heater GUI would consist of:

1 A temperature sensor, represented by a Tk widget and updated on periodic basis.
2 An alarm widget that is raised at a critical temperature level.
3 The operator of the system sets the reference temperature *r(t)* by means of a scale widget.

The set of *user interface primitives* used by an application should be provided in a package object segregated from the rest of the application. We shall refer name this object as `User_Interface` (Fig. 9).

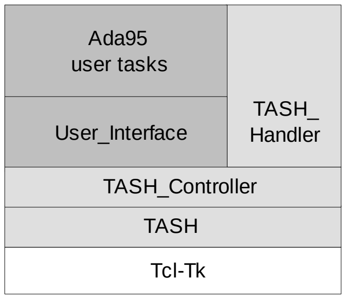

Fig. 9 The deferred server (III)

For example, the heater user interface primitives are `Update_Temp`, used to update the temperature widget, and `Show_Alarm`, used to trigger the alarm widget.

The `User_Interface` package would be:

```
with TASH_Controller;
package User_Interface is
  procedure Initialice_UI(...);
  procedure Update_Temp(...);
  procedure Show_Alarm(...);
end User_Interface;
```

Associated to the scale widget there is an Ada95 Application Specific Command, `Set_Ref_Cmd`, invoked by `Do_Events` on user scale events. `Set_Ref_Cmd` and, in general, any user Application Specific Command, have two fields of activity as Fig. 10 shows. First, the user code, to update the Ada95 real time variable that holds the reference temperature. The Application Specific Commands should be the only place where the operator updates global control variables of the system. Secondly, the Tcl-Tk library, which shows the new reference value on the screen. `Set_Ref_Cmd` also uses `TASH_ Controller`, as `User_Interface` entries do.

The question is where to implement the Application Specific Commands, such as `Set_Ref_Cmd`. The Ada95 Application Specific Commands are never called from any Ada95 code, but only from the Tcl-Tk library as callbacks. Thus, placing its interface in the specification of `User_Interface` is harmless, but useless. Our solution is to put them all in a separate package, `User_Commands` as `xx_Cmd` functions (Fig. 10).

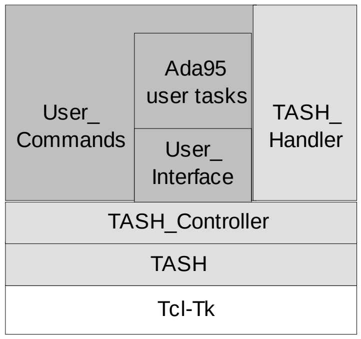

Fig. 10 The deferred server (IV)

Needless to say, the Tcl-Tk library has to know the address of each `xx_Cmd` function in order to invoke it. For example, the Ada95 code

```
cmd := CreateCommands.Tcl_CreateCommand(
  Interp, "decrease",
  Decrease_Cmd’access, 0, NULL);
```

shows how the main procedure passes the address of `Decrease_Cmd` to the Tcl-Tk library. `User_Commands` needs a method, `Create_User_Command`, that lets Tcl-Tk know about every `xx_Cmd` function and is invoked by `User_Interface` as part of its initialization. For the heater system, for example, the definition of `User_Commands` would be as follows:

```
with TASH_Controller;

package User_Commands is
  procedure Create_User_Command(
              Interp : Tcl_Interp);
end User_Commands;

package body User_Commands is
 package C renames Interfaces.C;
 package CreateCommands is new
  Tcl.Ada.Generic_Command(Integer);

 function Set_Ref_Cmd (
  ClientData : in Integer;
  Interp     : in Tcl_Interp;
  Argc       : in C.Int;
  Argv       : in CArgv.Chars_Ptr_Ptr
 )return C.Int;
 pragma Convention(C, Set_Ref_Cmd);

 function Set_Ref_Cmd(
  ClientData : in Integer;
  Interp     : in Tcl_Interp;
  Argc       : in C.Int;
  Argv       : in CArgv.Chars_Ptr_Ptr
 )returns C.int is
 begin
   -- 1. Update the Ada95 Reference
   --    Temperature global variable
   -- 2. Show the Reference in the
   --    screen via TASH
 end;

 procedure Create_User_Command(
             Interp : Tcl_Interp) is
 begin
  -- Create Ada95 App. Specific Commands
  -- One entry per User Extended Command
  CreateCommands.Tcl_CreateCommand(
    Interp, "Set_Reference",
    Set_Ref_Cmd’access, 0, NULL);
 end;
end User_Commands;
```

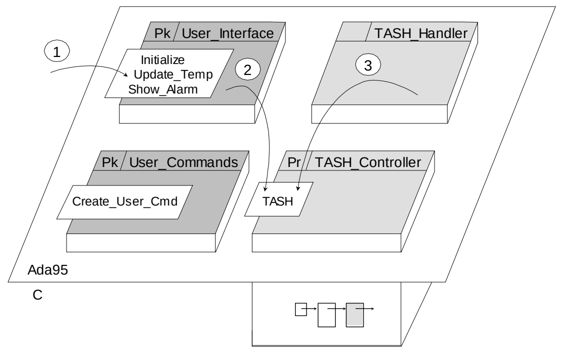

Fig. 11 First design attemp. Succsessful case

We can see a first working example (Fig. 11):

1. When an Ada95 task decides that the temperature widget must be updated, the `Update_Temp` user interface primitive is invoked.
2. `Update_Temp` uses the TASH interface encapsulated in the `TASH_Controller` to introduce an event in the queue (the operator does not notices any change on the screen).
3. `TASH_Handler` wakes up and invokes `Test_Event`, which causes the invocation of `Do_Event` to serve the event introduced in step 2, as expected. The new temperature value appears on the screen.

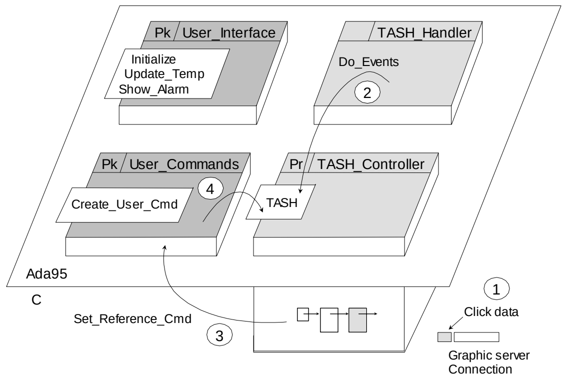

Fig. 12 First design attemp. Faulty case

Fig. 12 illustrates a second example where a user callback takes place:

1. The operator sets the reference temperature through the scale widget via mouse. The graphics library sends the data to the connection (again, the operator fails to notice any change on the screen).

2. Eventually, `TASH_Handler` wakes up and invokes `Test_Event`, which causes the invocation of `Do_Event` to read the connection, make an event and serve it.
3. To serve the event, `Set_Ref_Cmd` is invoked as an user callback.
4. When `Set_Ref_Cmd` tries to use TASH to put the value on the screen, it finds the `TASH_Controller` protected object closed by the current thread. A deadlock occurs. The first attempt has failed.

6.2 A second design attempt

One solution to the deadlock problem is given by Application Specific Commands in order to overcome the protection of `TASH_Controller` by directly invoking TASH. This impose providing a **double interface** for TASH, the protected one, `TASH_Controller`, and the unprotected, `TASH` (Fig. 13).

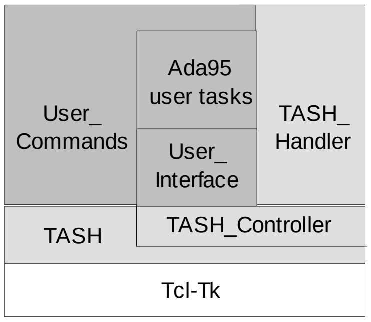

Fig. 13 The deferred server (V)

Under the double interface facility, step 4 does not cause occasion deadlock because, now, the thread goes only once through the protection of `TASH_Controller`. This is shown in Fig. 14.

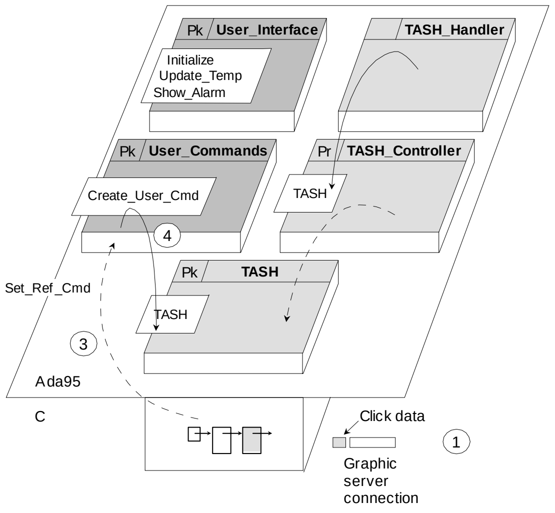

Fig. 14 Second design attemp

Although it seems clear that the double interface is not the ideal solution to the reentrance problem introduced by user callbacks, it also seems that there is not a better one. Notwithstanding, Ada95 applications are kept well structured because the solution exhibites a useful property: `User_Commands` only invokes the raw `TASH` interface, while `User_Interface` only invokes the protected `TASH_Controller`. Fig. 13 shows the definite deferred server architecture.

A well known case study, the Mine ontrol System ([2]), has been implemented with a Tcl-Tk GUI built upon the deferred server model. We have used Linux 2.0.X, GNAT 3.10p and Tcl-Tk 8.0. Fig. 15 shows a snapshot of the GUI.

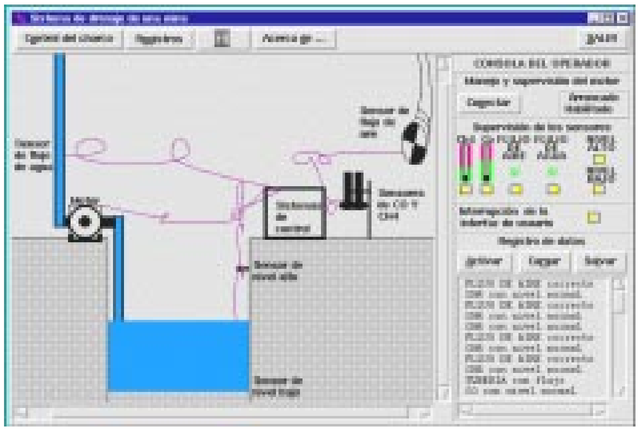

Fig. 15 The Mine Control System Tcl-Tk GUI

7. THE HRT-HOOD APPROACH TO THE DEFERRED SERVER

HRT-HOOD is a well known object based design metodology that helps to build reliable big real time systems ([1], [2]). Each HRT-HOOD terminal object has an automatic translation to an Ada95 package with timing attributes. The resulting running system is a concurrent set of Ada95 tasks. Non-terminal objects are called *active*, while there are four kinds of terminal objects: *Passive*, *protected*, *cyclic* and *sporadic*. As an example, the heater system of Fig. 8 is described in HRT-HOOD terms by Fig. 16. Its user interface has been encapsulated in the `Operator_Console` active object.

HRT-HOOD requires that the operations on an object delay the invoking thread for a bounded time only. Therefore, the Tcl-Tk written `Operator_Console` object enforces this rule with operations restricted to be ASER (Asynchronous Execution Request).

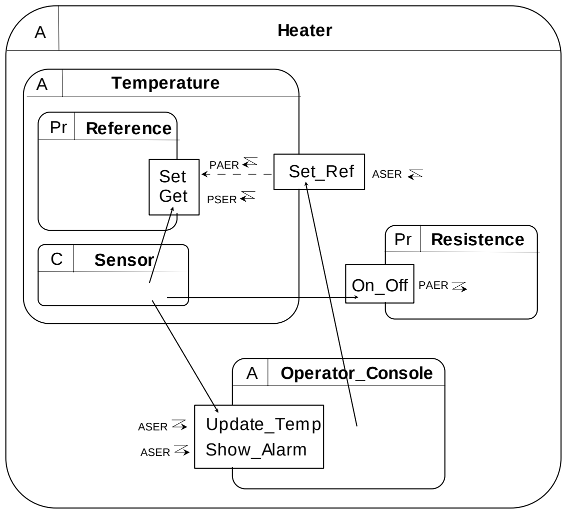

Fig. 16 A first HRT-HOOD design of the heater system

Therefore, general purpose third-party libraries, such as Tcl-Tk, are not prohibited components in the implementation of any object. What is relevant is to enforce the HRT-HOOD design restrictions, such as syncronization or timing. The deferred server is a method of encapsulating Tcl-Tk in Ada95 objetcs that can be used by a concurrent application. The question, therefore, is whether, without loss of generality, the deferred server architecture can be used to implement the HRT-HOOD `Operator_Console` object of Fig. 16. In the afirmative case, every HRT-HOOD system using a deferred server Tcl-Tk user interface would match Fig. 17.

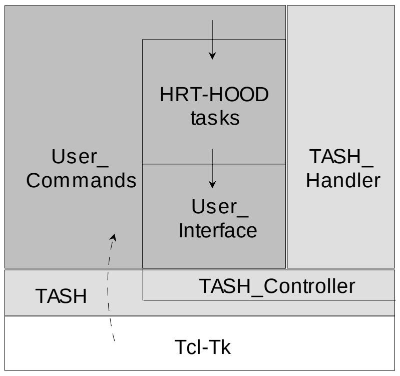

Fig. 17 Hipothetical HRT-HOOD system based on the deferred server architecture

The implementaton of Fig. 17 raises the operating system problem. To achieve predictability, the HRT-HOOD tasks should stand alone, without the operating system support. Tcl-Tk, however, needs an X-Windows and a Unix box. In order to guarantee predictability to the real time tasks, Fig. 18 shows a distributed configuration, where the X-Windows server is moved to a different dedicated machine or X-terminal. Then, the operating system disappears from the real-time system and the network adapter is controlled by HRT-HOOD objects implemented conforming to the low level programming Ada95 facilities ([2]).

The main shortcoming of this approach is that Tcl-Tk and the X client library must be modified in order to support the operating system services by themselves. This possibility has been explored. The single line Tcl script

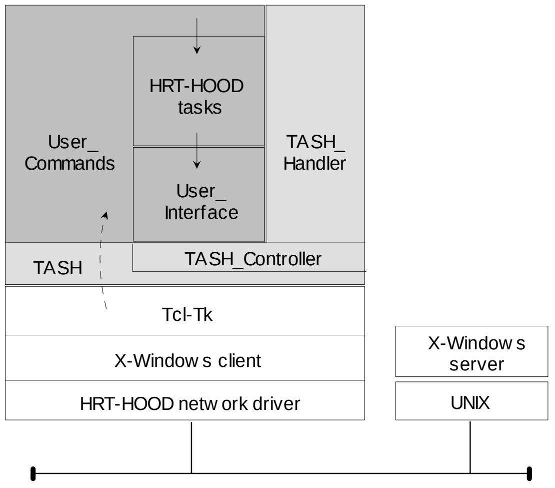

Fig. 18 Distributed HRT-HOOD/Tcl-Tk system (I)

```
#!/usr/bin/tclsh
puts stdout "Hello, world"
```

makes 18 different system calls and a total number of 86. The 26 lines Tcl-Tk script `Adding.tcl` that builds the GUI in Fig. 2 makes 25 different system calls and a total number of 488. It is true that the system is now fully analizable in its temporal requirements, but Tcl-Tk and the client component of X-Windows entrust functionality to the operating system. This is difficult to be assumed by themselves and the Ada95 run-time system.

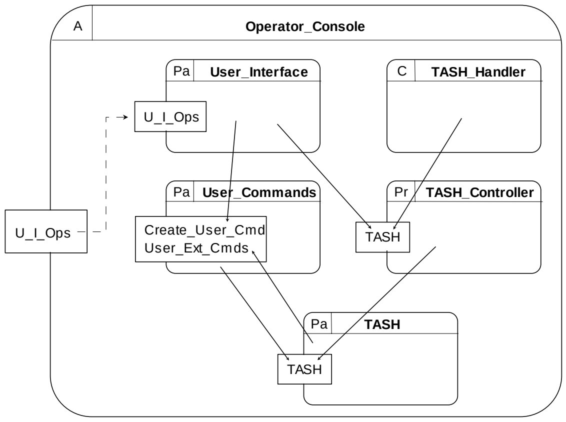

Fig. 19 A faulty HRT-HOOD design of the Tcl-Tk User Interface

The operating system problem, however, is not the only one. Fig. 19 is a refinement of `Operator_Console` into terminal objects. As it can be infered by the object names and relationships, the goal of this decomposition is to provide an automatic mapping of the Ada95 deferred server architecture proposed in previous sections of this study. Some design and implementation problems can be hence identified:

First, HRT-HOOD terminal objets must be temporally analizable. An HRT-HOOD protected object has two attributes, the ceiling priority and the worst case execution time (WCET). The latter refers to its slower operation and determines the blocking imposed on the invoking tasks. Since the HRT-HOOD protected object `TASH_Controller` is implemented as the Tcl-Tk library, to which it guarantees mutual exclusion, it is quite difficult to determine its WCET parameter, due to the unpredictable operating system time response.

Secondly, HRT-HOOD specification ([1], pg. 29) explicitly "forbids passive (or protected) objects to use each other in a cyclic manner". Tcl-Tk user callbacks necessarily introduce a cycle between `User_Commands` and `TASH`.

Thirdly, HRT-HOOD specification ([1], pg. 30) explicitly establishes that passive objects can only invoke operations on passive objects. Passive `User_Interface` invokes protected `TASH_Controller`.

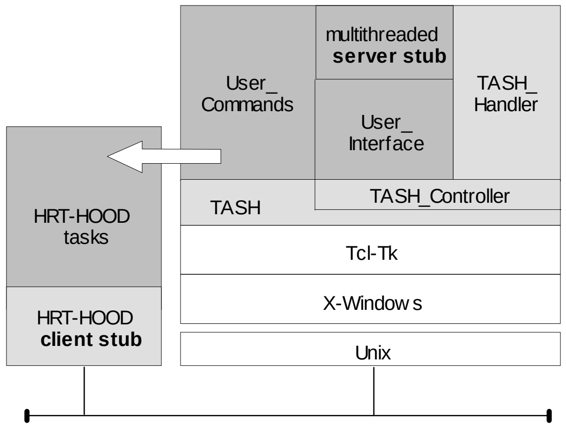

Fig. 20 Distributed HRT-HOOD/Tcl-Tk system (II)

The run-time analysis and, more important, the inherent violations of HRT-HOOD rules described above make clear that to embed Tcl-Tk in a HRT-HOOD system is not possible. Therefore, other alternatives must be explored. Fig. 20 shows a distributed configuration where the deferred server goes to a second machine, as a remote service to the HRT-HOOD system. The interface definition of this service consists of the user interface primitives of the real-time system. As Fig. 20 shows, the HRT-HOOD tasks are replaced in the server side by a multithreaded server stub.

User callbacks are problematic. We may ask what happens when a user extended command accesses a real-time variable. Now the HRT-HOOD system is not only a *client* of Tcl-Tk services, but it becomes a *server* of its real-time variables, monitored and controlled by the operator. Notwithstanding, the `Operator_Console` object becomes much lighter that the one in Fig. 19. It implements the client stub by taking control of the network adapter. Its operations are ASER in order to bind the delay imposed to real-time tasks: They return when the packet has been inserted in the network. It also implements the real-time variables service stub by introducing a thread that listens for real time requests in the network adapter. As an example, Fig. 21 shows the operator console object for the heater system.

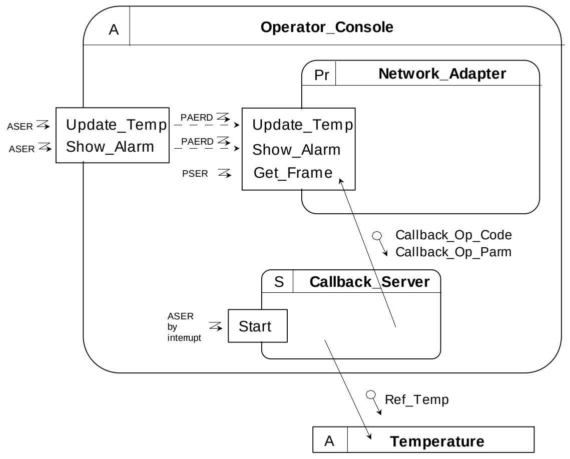

Fig. 21 A second HRT-HOOD design of the Tcl-Tk User Interface

8. WORK UNDER WAY

The deferred server architecture has been implemented and tested in a centralized way; not yet as a remote service of a HRT-HOOD system. The main problem to be solved is the communication between the two machines in Fig. 20. Three possibilities are explored. First, to build an analizable TCP/IP HRT-HOOD active object in the real time side. We plan to model it after the Minix user space TCP/IP server process. One second possibility is to unload the real time side from any communication protocol. Assuming the system shown in Fig. 20 is a reliable local network, the big and heavy network and transport protocols seem unnecessary. TCP/IP is, however, a good choice in the GUI side because it allows Tcl-Tk programming on a full standard graphic platform, such as X-Windows. The drawback of this approach is that it forces the real time side to fake the GUI side by inserting the TCP/IP stuff in the outgoing frames and discarding it in the incoming ones. Finally, a very attractive option is to explore a much ligther network protocol than TCP/IP, the network protocol of the Amoeba operating system: FLIP ([6]).

9. CONCLUSIONS

First, the problem of using the Tcl/Tk scripting language in mutithreaded Ada95 applications has been studied, while a solution proposed: the deferred server architecture. This demands a little extension to Tcl-Tk/TASH. A complex enough classic example, the Mine Control System, has been implemented to test these ideas with success. Secondly, the integration of Tcl/Tk in HRT-HOOD systems has been explored. The reentrant nature of Tcl-Tk and its mechanism of callbacks allow us takes us to conclude that is not possible to build a stand alone true hard HRT-HOOD system with a Tcl-Tk user interface. Finally, in contrast, a distributed approach to the Tcl-Tk GUI that relies on the deferred server model, seems like a promising option. This distributed architecture has been described, and its problems pinpointed. posed. Work is in progress in order to gain experience on this approach; we aim to achieve the goal of developping true HRT-HOOD systems with Tcl-Tk script based user interfaces.

10. REFERENCES

[1] Burns, A. and Wellings, A., *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems*, Elsevier, 1995.

[2] Burns, A. and Wellings, A., *Real-Time Systems and Programing Languages*, Addison-Wesley, 1996.

[3] Díaz Martín, J.C., Irala Veloso, I., "Prácticas de Sistemas de Tiempo Real en la Uex. Integración de Tcl-Tk en HRT-HOOD", *Jenui'98, Actas del congreso, pp. 166-172, Escola d'Informática d'Andorra. Sant Juliá de Lória, Andorra*, July, 9-10, 1998

[4] IEEE, "Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) [C Language], IEEE Std 1003.1, 1996 Edition, (1996).

[5] Ousterhout, John, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA, 1994.

[6] Tanenbaum, A. S., *Distributed Operating Systems*, Prentice-Hall, 1995.

[7] Welch, B., *Practical Programming in Tcl & Tk*, Prentice-Hall, 1997.

[8] Westley, T., "TASH: Tcl Ada Shell, An Ada/Tcl Binding", *ACM SIG Ada Ada Letters*, 1996.