Software for Providing Remote User Interfaces— the RTC Library

First Revision, printed June 17, 1997
R. J. Kirkham

Commonwealth Scientific and Industrial Research Organisation
Division of Manufacturing Technology
Melbourne Laboratory

Corner of Albert and Raglan Streets, Preston, Victoria
Postal Address: Locked Bag 9, Preston 3072 AUSTRALIA
Telephone: +61–3–9662–7700 Facsimile: +61–3–9662–7851
World Wide Web: `http://www.mlb.dmt.csiro.au/`

Document History

June 17, 1997	First Revision

Summary

This report describes RTC, which stands for Remote Tool Control. RTC is an integration of an RPC (Remote Procedure Call) network protocol with the Tcl (Tool Control Language) programming system [4, 8]. It is intended to facilitate the writing of graphical user interfaces to control systems remotely through a computer network.

RTC uses a somewhat different approach to producing graphical interfaces to that used in the past within the Division's Industrial Automation Programme. Firstly, the interface itself uses Tcl and Tk (Tcl's graphical interface toolkit) [8]. Tcl/Tk permits very fast development of graphical interfaces, as a given interface can be implemented with much less code compared to the traditional C approach. As the language is interpreted, the development/test cycle is faster because there are no compilation delays.

Secondly, the underlying RPC protocol used to communicate with the remote computer is fixed. RPC is a good choice, but in the past a new RPC protocol was developed for each application. This meant that new 'glue' code between the protocol and the application has to be re-written each time. By fixing the RPC protocol, both the client and the server can be generic, largely pre-written and available in the RTC library, eliminating much application-specific C code.

The remote procedure call protocol RTC provides is modelled after the Tcl function call: the function accepts a number of string arguments, and produces a string result. This protocol has been found to have considerable advantages, and integrates smoothly with both Tcl clients and C servers.

The RTC system provides a Tcl run-time loadable module, used in Tcl RTC clients, and a library for linking with RTC clients and servers written in C. There are also some diagnostic programs useful for interface development. The system, or appropriate parts thereof, has been tested on Solaris, SunOS, LynxOS, Linux, and *vxWorks* operating systems.

Following a short introduction to remote user interfaces in general, this report describes how to use RTC, through simple examples and descriptions of the Tcl procedures, C library functions, and diagnostic programs. This is followed by a section dealing more specifically with Tcl user interfaces using RTC, including a case study of the PIRAT user interface [1, 7]. A final section describes how to install the RTC 'distribution'.

Contents

1 User Interfaces for Remote Systems

1.1 Introduction

Several projects conducted within the Division's Melbourne Laboratory in the last few years have included a graphical user interface to control a remote system. The interface, generally an X Windows program written in C, runs on a UNIX workstation and uses an application-specific RPC (Remote Procedure Call) protocol to send commands to and receive data from the remote computer through a network. In other words, the interface runs on one computer, the program or system it controls on another, and the two communicate using a custom protocol.

For example, the Buderim Ginger system featured an elaborate XView user interface running on a Sun workstation, which communicated using two linked RPC protocols with the MVS (Machine Vision System) processor [5, 6]. The interface controlled and monitored the MVS, displayed its state to the operator, flashed up the MVS's alarm messages, and at the same time allowed the operator to edit the ginger sorting programs.

These interfaces use RPC not because the remote system is actually physically distant (although it can be), but because the remote system does not provide a suitable graphical user interface infrastructure. The VMEbus microprocessor modules frequently used for real-time systems (such as the MVS) rarely have a video adaptor to use for an interface, and if they did, it would mean porting a great deal of graphical user interface code to the real-time kernels generally used on such hardware.

These interfaces are *graphical* because of the perceived need to surface prototypes with a fancy, colourful veneer—to show they are 'hi-tech'. This seems to occur even when such an interface is not completely appropriate, such as with the Buderim system. There, the factory-floor operators have to enter the air-conditioned room where the Sun workstation is kept, remove their rubber gloves, and fiddle with the interface to enter two pieces of information, before they can actually turn the machine on.

Leaving aside the question of appropriateness, the author has never been very convinced of the efficiency of the current method of constructing such interfaces—`rpcgen` and the XView toolkit—for reasons described shortly [4, 3]. Over the past few years he has sporadically tinkered with various other approaches, before the imperatives of the PIRAT project led him to implement RTC the remote interface system that is the subject of this report [1, 7]. Before describing RTC however, it seems worthwhile to examine some general matters related to remote user interfaces, starting with the apropriateness issue mentioned above.

1.2 Why have a Remote Interface?

A variety of reasons are proffered for having remote graphical interface. Not all of these are good reasons. A few are examined below, along with some counter-arguments:

The remote system is indeed remote. It's hard to argue with this, of course. An excellent example of this situation is the Safe-T-Cam system, where computers at unmanned sites around New South Wales are linked by an ISDN network to a centre in Sydney [?]. Each site can be represented by an interface window on an operator's screen and be conveniently interrogated and commanded through it. However, this is not necessarily an argument for a *graphical* user interface: see the points below.

The computer has no display hardware. The VMEbus computer hardware usually often at the heart of high-performance industrial real-time systems is notorious for providing no user interface hardware beyond a serial port, so networking it with another computer that *does* have a display seems an obvious step. However, since these computer modules are usually there to control application-specific hardware, it is not clear why adding some relatively simple hardware to provide a user interface is apparently overlooked. Section **??** describes local interface options further.

The operating system does not support a display. This tends to be the case with the shared-memory real-time kernels, such as *vxWorks* and RTEMS [9, 2]. To an extent this relates to the preceeding point, although these system's philosophy is often to provide only basic multi-tasking and IO facilities, and let the programmer write or obtain software for this application-specific feature.

They unify a disparate system. A complicated system which involves a number of computers, even if not remote, might use remote interfaces to collect the controls for each node onto a workstation screen for the benefit of a single operator. For instacnce, it was the main reason for choosing to use remote interfaces in the PIRAT system.

They make the system easy to use. This is often asserted in favour of graphical user interfaces in general, but is of course only true for a *good* graphical user interface. In fact, a good design will have distilled the interface/operator interactions down to the optimum set required for effective control of the system. This set is independent of the interface *technology*. In other words, a well thought-out interface constructed using old-fashioned electrical buttons and lamps might well be easier to use than a hastily assembled graphical interface.

They look good. This argument is typically allied with the preceeding one, in the context of providing a 'professional' looking system. Clearly a completed system should be aesthetically pleasing, but in the author's view the aesthetic extends beyond a colourful display to engineering design issues such as its appropriateness to purpose.

1.3 Why *not* have a Remote Interface?

As well as good (and not so good) arguments for having remote graphical interfaces, there are good arguments *against* having them:

The users might not want one. An excellent example is the Buderim operators with their rubber gloves. While graphical interfaces are becoming widespread, they are not yet common on factory automation, and may be unfamiliar or inconvenient for many people. Clearly the design and technology of the interface must consider its final users.

The software structure is more complicated. A remote interface complicates the structure of a system, because a network communication protocol must be devised, and code written to interface it to both the main application and its remote interface. This means the system will contain more code than if the interface had been integrated into the main application, assuming this is possible: from this it follows that the system will take longer to design, implement and test.

They show up unreliability. Two separate programs must cooperate effectively over a network. Any unreliability in either computer or the network between them will be reflected at the most visible point—the user interface. Designing software to cope with unreliability in a safe and convenient manner is not always easy. And while the user may be merely inconvenienced by the lock-up of a graphical interface, someone else may be placed in danger by machinery under control of the failed system.

They might be insecure. Most networks are subject to an unauthorised access risk to some degree. The dangers of placing control of a system in a network protocol must therefore be carefully considered from that point of view as well.

They mean another computer has to be there. Where the remote interface is not actually remote, it means there are two computers in proximity, with presumeably twice (or whatever) the capital and maintenance burden. This may be significant, especially if the preferred platform for the graphical interfaces a Sun Workstation.

2 Using RTC: An Example

An interface using RTC can be developed server-first, client-first, or both can be worked on together. It is probably more straightforward to establish the server apparatus first, test it, and then work on the client side.

To illustrate using RTC, an example application is presented here. It is very simple indeed—and correspondingly useless—but can be tried to make sure you can compile and link servers, and use `wish` with RTC. For the sake of brevity, the example lacks some of the error checking code that would normally be included on both the Tcl and C side. The source of the example programs can be found in the RTC distribution.

2.1 User Protocol Definition

Let us assume that we want a graphical interface (client) for printing messages on one of a number of serial terminals connected to a remote computer (server). The client might have to work with a couple of different servers, which might have different numbers of serial ports.

This suggests the server should provide two remote functions: one to report the names of the available serial port, and the other to print a message on a given serial port. The client should use the first function to present a list of ports to the user to select from, then accept a message from the user, and use the second function to print it.

2.2 Server Side Implementation

The server side in the RTC system is always written in C. It comprises a number of *stub functions* (in this case, two), and a `main()` function which:

- *initialises* the server part of the RTC library
- *registers* the stub functions with the server, and
- *enters* the server wait-loop itself.

The server skeleton looks like this:

```
#include "rtc.h"

char *listports(int argc, char *argv[]) {
    return "OK";
}

char *message(int argc, char *argv[]) {
    return "OK";
}

int main(int argc, char *argv[]) {

    /* initialise RTC server library */
    rtcServerInit();

    /* register server stubs */
    rtcServerRegister("message", message);
```

```
        rtcServerRegister("listports", listports);

        /* enter server loop */
        rtcServer("udp", 33, argv[0]);
    }
```

The two stub functions are called `listports()` and `message()`. Note that the arguments are strings, passed using the `argc/argv` convention, just like function `main()`, but unlike `main()` the function must return a string pointer result.

The server guarantees that the stubs are always called with at least one argument: `argv[0]` is the Tcl name the function was invoked by, and `argc` is always 1 or greater. Real arguments begin at `argv[1]`. The server also guarantees that `argv[argc]` is always `NULL`.

The `main()` function first initialises the server part of RTC library, and then registers our tow stub functions, using `rtcServerRegister()`. The first argument is the *name* by which the function will be known, and the second is a pointer to the function.

If desired, a stub function can be registered under more than one name (the stub can use `argv[0]` to tell what name it was invoked under). Stubs can also be re-registered and de-registered, even after the server has started. This permits 'dynamic protocols' to be implemented, something that is not possible using traditional static RPC.

Finally `main()` enters the server loop, from which it should never exit. The server accepts a transport type argument (here, the User Datagram Protocol UDP has been selected), a *program number* (more about this later), and a string used to identify the server in any log messages it may produce.

2.3 Implementing the Rest of the Server

Now that the skeleton of the server is in place, the server stubs can be fleshed out (alternatively, you may wish to try to compile and test the server as it stands, as described in the following section—it should work).

The simpler of the two stubs is `message()`. We will make the first argument to this function be the name of the serial port, and the second the message string to write to it:

```
    #include <stdio.h>

    char *message(int argc, char *argv[]) {
        FILE *port;

        /* open the serial device */
        if (! (port = fopen(argv[1], "w")))
            return "ERROR Can't open port";

        /* print out message, close port */
        fprintf(port, "message: %s\n", argv[2]);
        fclose(port);

        /* return ok result */
        return "OK";
    }
```

The function returns a string indicating its success or failure. It is up to the user client software, which calls this function through the RTC mechanism, to interpret the result, and take action: RTC does not interpret the return strings in any way, and does not care if a stub function thinks it has failed.

The `listports()` stub is a bit more complicated. It ignores any arguments, but returns a quite long string containing a space-separated list of the filenames of all the serial port devices it can find. It assumes that any file in the `/dev` directory with a name beginning with `tty` is a serial port, and uses the UNIX directory-searching functions to compile the list:

```
#include <fcntl.h>
#include <string.h>
#include <dirent.h>

char *listports(int argc, char *argv[]) {
    DIR *dirp;
    struct dirent *dirent;
    int length = 100;
    char *result = rtcServerBuffer(length);

    /* open directory */
    if (! (dirp = opendir("/dev")))
        return "ERROR Can't open directory";

    /* go through each directory entry */
    while (dirent = readdir(dirp))

        /* accept only names that start with "tty" */
        if (strncmp(dirent->d_name, "tty", 3) == 0) {

            /* extend result string as required */
            if (strlen(result) + strlen(dirent->d_name) + 8 >= length) {
                length += 100;
                result = rtcServerBuffer(length);
            }

            /* append to result */
            strcat(result, " /dev/");
            strcat(result, dirent->d_name);
        }

    closedir(dirp);

    return result;
}
```

Up until now, we have only seen stub functions that return static strings as results. However, they may also return one of their own arguments (or a sub-string of an argument), or a dynamically allocated string obtained through the function `rtcServerBuffer()`.

This function is used by stubs to get working memory for buffers or for returning results, especially when the length of the result isn't known in advance, as in the example. You should use it instead of `malloc()` and `realloc()` because the memory allocated by `rtcServerBuffer()` is automatically reclaimed by the RTC server after the stub function returns.

The first time `rtcServerBuffer()` is called, it returns a pointer to a guaranteed clear buffer of at least the requested size. Subsequent calls re-size this buffer, larger or smaller: if necessary, it moves its contents to a new location. This feature is exploited in `listports()` above; as the length of the list of ports gets longer, the buffer containing it is expanded (in lots or 100 bytes, for the sake of efficiency).

Since this function has to return a list, it might be argued that the result should be returned in an `argc/argv[]`-format as well. However, this makes the stubs more complicated, and besides Tcl is very good at dealing with space-separated tokens—this is what a Tcl list *is*. So, Tcl functions such as `lindex` and constructs such as `foreach` can be used to deal with these sorts of results.

2.4 Compiling and Testing the Server

Assuming all this server code is in the file `server.c`, you can compile and link the server:

```
magni% gcc -o server server.c -lrtc-sparc-solaris
```

You will probably need some `-I` and `-L` compiler options so it can locate the RTC header file `rtc.h` and server library `librtc-sparc-solaris.a`, and on operating systems other than Solaris, different libraries. Once compiled, the server can be run (do this on a separate window, if possible):

```
magni% server &
[1] 15259
magni%
```

The trailing `&` puts the server process in the background. Now, the server code can be tested, using the `rtcclnt` program, part of the RTC package:

```
magni% rtcclnt
This is an RTC process
Library timestamp 970304174233
magni/0>
```

`rtcclnt` is a command-shell RTC client program which allows the user to interrogate and test RTC servers (without actually using Tcl: see Section 4.2).

There are commands to set client parameters, and perform remote calls and view the results (the command `help` prints a menu). Firstly, we have to use the `program` command to set the program number to 33, which is the number our server was registered with. (There is nothing special about 33, it's just the number being used for this example). Then, we can try a remote call on our server, using the `call` command:

```
magni/0> program 33
magni/33> call catalogue
listports message ping catalogue
magni/33>
```

The remote call we tried was called `catalogue`. RTC servers always have two in-built remote functions, `ping` and `catalogue`. `ping` always returns the name by which it was invoked (usually, `"ping"`), and `catalogue` returns a space-separated list of the names of all the remote functions registered with the server. Note that our two server stubs, `listports` and `message`, appear here.

Now we can try calling one of these ... the moment of truth!

```
magni/33> call listports
/dev/tty /dev/ttyp0 /dev/ttyp1 /dev/ttyp2 /dev/ttyp3 /dev/tt
yp4 /dev/ttyp5 /dev/ttyp6 /dev/ttyp7 /dev/ttyp8 /dev/ttya /d
ev/ttyb
magni/33>
```

The list has wrapped around the screen, but this does not matter: our function seems to work. We can test the `message` function as well:

```
magni/33> call message /dev/tty Hello!
OK
magni/33>
```

Over on the window where the server was running, we should see the output:

```
message: Hello!
```

2.5 Tcl Client Side

Having proved the server is functioning correctly, development of the Tcl client can commence. Using RTC in a Tcl ot Tcl/Tk application script is a three-stage process. The script must:

- load the RTC loadable object module
- obtain and configure an RTC *handle*
- use the handle to perform remote procedure calls.

RTC is implemented as a loadable object module, which can be loaded into a running Tcl interpreter using the `load` command, which is available in Tcl versions 7.5 and later. (Previously, the RTC routines were statically linked with Tcl and Tcl/Tk to produce the special interpreters `rtclsh` and `rwish`).

While loadable modules solve the considerable problem of multiple, incompatible versions of Tcl and Tcl/Tk interpreters each with different Tcl extensions built in, they introduce the lesser problem of locating the correct loadable module for the binary architecture of the computer. At present, the best solution to this is not yet apparent, and for our example we will use the following, based on the `platform` script:

```
# load the RTC module
set RTC /opt/rtc
load $RTC/lib/rtc-[ exec platform ].so Rtc
```

The RTC module adds one new command to the Tcl or Tcl/Tk interpreter: logically enough, it is called `rtc`. The command `rtc` creates an RTC *handle*—in fact, a new Tcl command—which represents a particular server. The name of the handle is the first argument to `rtc` (similar to the Tk *class commands*). Optional arguments specify the hostname the server is on, its program number, time-out periods, and so on:

```
# create an RTC handle
rtc remote -server magni -transport udp -program 33
```

This has created a new Tcl command called `remote`, representing a RTC server providing program 33 running on host `magni`—in other words, matching our example server above. The `rtc` command does not actually communicate with the server, or use RPC at all; this only happens when the handle is actually used.

You can have as many handles as you wish, representing different servers, or even the same server. The only requirement is that the handles have different names (and are different from any existing Tcl function names).

The handle is invoked with one of three action arguments: `call`, `configure`, and `destroy`. To actually do an RTC call to our server, use `call`:

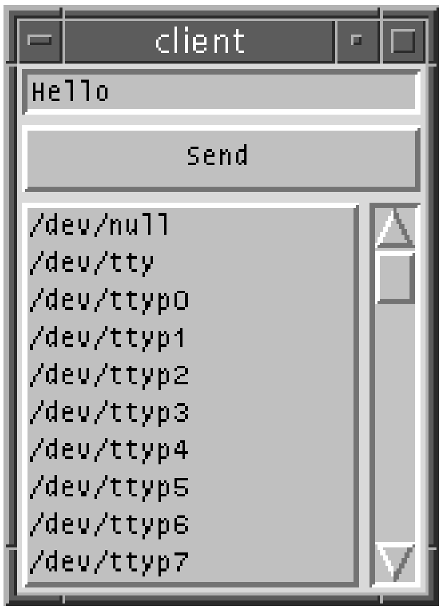
```
# get the 3rd port from the server and say hello to it
set ports [remote call listports]
remote call message [lindex ports 2] Hello!
```

This calls our server's `listports` remote procedure, and stores the list in the `ports` variable. This will be a list variable, so the Tcl `lindex` command can be used to extract simple items from it. Assuming we want to say hello to the third port on the list (which happens to be `/dev/ttyp1`), the second line calls the `message` remote procedure, passing the selected file name as the first argument, and `Hello!` as the second. If all goes well, a message should appear on the server on `/dev/ttyp1`.

The RTC handle's configuration can be changed at any time using the `configure` action argument; `configure` accepts the same options and arguments as the `rtc` command that created it. Once the script has finished using the remote server, it can invoke the handle once more, but this time with the `destroy` action argument, which, as might be expected, closes any connections with the server and reclaims the resources used by the handle.

2.6 Completing the Tcl Client

Finally, having shown we can call a remote server from Tcl, we can complete our simple serial port message application with a reasonably glossy Tcl/Tk graphical user interface. To operate, the user types an message in the box at the top, selects the port they want from the list, and presses Send:



```
Hello
/dev/null
/dev/tty
/dev/ttyp0
/dev/ttyp1
/dev/ttyp2
/dev/ttyp3
/dev/ttyp4
/dev/ttyp5
/dev/ttyp6
/dev/ttyp7
```

The Tcl script that produced this interface is below. It begins as a standard shell script, but with a cunning prologue that re-executes itself as the standard `wish` Tcl/Tk interpreter, with the rest of the script as source. The script then loads RTC and creates an RTC handle as demonstrated above.

Then follows a sequence of standard Tk commands to create the user interface. There is an entry box for the message, a Send button, and a scrollable list box for the available serial ports. The first RTC call, which gets the list of ports from the remote server, occurs just after the initialisation of this list box. The names of the ports are `insert`ed one by one into the list using a `foreach` loop. Note that an initial entry for `/dev/null` is inserted beforehand; this is so the list will have at least one sensible entry, in case the server does not report any available ports. The list box is in single-selection mode, so only one serial port can be selected at once.

The second RTC call occurs only when the user presses the Send button, because the call is part of the button's `-command` option. The selected serial port name comes from the list box (a two-stage process), and the string to print from a variable controlled by the entry box.

```
#!/bin/sh
# the next line restarts using wish \
exec wish "$0" "$@"

# load the RTC module
set RTC /opt/rtc
load $RTC/lib/rtc-[ exec platform ].so Rtc

# create an RTC handle called remote
rtc remote -program 33

# set up message entry and send button
set entry Hello
entry .entry -textvariable entry
pack .entry -side top -fill x

button .send -text "Send" -command {
    remote call message [ .list get [.list curselection]] $entry
}
pack .send -fill x

# set up scrolled listbox of available ports
listbox .list -selectmode single -width 20 -relief raised \
    -yscrollcommand ".scroll set"
pack .list -side left
scrollbar .scroll -command ".list yview"
pack .scroll -side right -fill y
.list insert end /dev/null
.list selection set 0

# get the ports from the server, insert into list
set list [ remote call listports ]
foreach port $list  {
.list insert end $port
}
```

2.7 Further Examples

Hopefully, this simple example has shown how easy it is to write RTC servers in C, test them out, and then write Tcl/Tk RTC clients to use them.

The example has a number of shortcomings, which the careful reader may have already noticed. In the Tcl/Tk client, there is no way to specify the remote server (by default it is the local host), the network transport to use (the default is UDP), or any program number other than 33.

There is also no handling of the error strings the remote calls might send back, or the Tcl exceptions that might be raised should the RTC/RPC mechanism fail somehow. More subtle is a problem in the server: the `message()` stub might block indefinitly writing to a busy serial device, which will freeze the interface until the RTC call times out.

So, while using RTC is quite easy, there are pitfalls as well. Section 5 of this report attempts to alert the reader to some pitfalls that have already been fallen into.

3 Library Function Descriptions

This section details the C and Tcl functions provided by the RTC library.

To use the RTC library from a C program, you must include the header file `rtc.h`. This file contains ANSI prototypes of the C functions, manifast contants, and macros. You must link the program against the library `librtc-<cpu>-<os>.a`, where `<cpu>` and `<os>` represent the CPU and operating system names, as revealed by the `platform` script. The library contains both the RTC server and client functions.

To use RTC in a Tcl script, you must first load the shared object module `rtc-<cpu>-<os>.so`.

Most of the RTC C functions return an integer result, zero on success, or non-zero on failure. In the latter case the function `rtcMessageGet()` can be called to return a string describing the error. Refer to Section 3.4 for details.

3.1 Server Side C Functions

`RTC_STATUS rtcServerInit(void)`

This function must be called before any of the other RTC server functions, because it initialises the RTC server data structures. It also registers the two standard RTC remote functions, `ping` and `catalogue`.

`RTC_STATUS rtcServerAuth(RTC_AUTH tag, ...)`

This function is called to add further entries to the server access control lists described in Section 5.11. The arguments are a set of one or more tag/argument pairs, terminated by a zero. The tags and argument types are:

Tag	Argument Type	Argument
`rtc_auth_host`	`char*`	client hostname
`rtc_auth_uid`	`int`	UNIX user-id number
`rtc_auth_gid`	`int`	UNIX group-id number
`rtc_auth_user`	`char*`	UNIX user name
`rtc_auth_group`	`char*`	UNIX group name

Note that the same tag may appear more than once within the same call. String data is duplicated by the function, so string arguments need not be static.

The authorisation lists initially contain the hostname of the server machine, and the user-id and group-ids of the server process. There is no way to remove an entry from the lists; in practice this has not been found to be necessary. `rtcServerAuth()` can be called at any time, even after the server has been entered.

`RTC_STATUS rtcServerRegister(char *name, RTC_EXEC function)`

Register the C server stub function `function()` with the server under the name `name`. The function will subsequently be invoked by the server task or process when a request to execute function `name` comes from a remote client, and it passes the autorisation check. The stub function must conform to the prototype:

```
typedef char* (* RTC_EXEC)(int argc, char *argv[]);
```

In other words, the function must accept `argc/argv[]`-style arguments, in the same way as the normal C `main()` function, but should return a character pointer. This must point to either static storage, to one of its `argv` strings (or a sub-string), or to dynamic storage obtained from `rtcServerBuffer()` below.

Only one stub function can be registered with a given `name`: subsequent registrations replace earlier ones. However, a stub function can be registered under many different names. Functions can be de-registered by passing `NULL` for the `function` argument.

`char* rtcServerBuffer(unsigned size)`

Server stub functions can call `rtcServerBuffer()` to get a pointer to dynamic memory for return values or scratchpads. The size of the buffer is given by `size`, and is guaranteed to be zeroed. The buffer is automatically de-allocated by the server after the call to the stub function is complete and the string has been transmitted back to the client. This permits stub functions to return large or variable size strings—and remain re-entrant—without fear of memory 'leaks'.

The buffer can be re-sized simply by calling `rtcServerBuffer()` with the new `size`: this is equivalent to `realloc()`. Note that the location of the buffer may change, and the function returns a new pointer. In this case the contents will be copied, so any pointers into the buffer will need to be re-computed.

`RTC_STATUS rtcServer(char *transp, unsigned prog, char *name)`

Establish an RTC server, and enter the server main loop, and do not return. In single-threaded UNIX systems, this is called at the end of `main()` after the server authorisation parameters have been set up using `rtcServerAuth()`, and the server stubs have been registered using `rtcServerRegister()`. In multi-threaded UNIX systems, and shared-memory multi-tasking systems, a new thread or task is generally established, which calls `rtcServer()`; `rtcServerAuth()` and `rtcServerRegister()` can be called before or after. Section 5.10 deals with multi-threaded environments in more detail.

Parameter `transp` is a string indicating the network transport type or types the server should provide. Generally this will be of the form `"tcp"`, `"udp"` or `"tcp udp"`, although some systems may provide additional or alternate transport types. Refer to Section 5.5 for a discussion of transport types.

Parameter `prog` is the RTC program number, given to this user protocol. Clients use the program number to find the server they want. Only one server per computer should have a given program number Section 5.9 deals with program numbers in more detail. Parameter `name` is a name for the server messages (`argv[0]` is normally used).

`RTC_STATUS rtcServerShow(void)`

Prints out a short summary of the server data structures, including the contents of the server authorisation lists and the names and addresses of all registered server stubs.

`RTC_STATUS rtcServerDone(void)`

This function deallocates the server data structures. On some systems it may attempt to remove the server thread or task. On single-threaded UNIX systems, there is no place to actually call this function from, but on these systems there is no particular need to, the the system will reclaim the resources itself.

3.2 Client Side Tcl Functions

The RTC Tcl interface—the client side of the system—has been modelled on the object-oriented style used by the Tk widget functions. In that arrangement, new *widget commands* with names given by the programmer are created using Tk's *class command* functions (such as `button` or `scrollbar`). Most of the parameters of the widget are specified by options to the class command. The widget is subsequently controlled by invoking the widget command with one of a set of action arguments (such as `configure`, `call` or `destroy`) and associated arguments.

`rtc` *`<handle>`* `[-server` *`<server>`*`] [-program` *`<program>`*`]`
`[-transport` *`<transport>`*`] [-timeout` *`<timeout>`*`] [-retry` *`<retry>`*`]`
`[-recreate] [-norecreate] [-debug] [-nodebug]`

Create an RTC handle with name ***`handle`***. ***`handle`*** can subsequently be called as a Tcl function to invoking remote function calls *`<handle>`* is the only mandatory argument to `rtc`: all options have useful default arguments. The options are described below. If an option is repeated, the last one prevails:

`-server` *`<server>`* (default `localhost`)
Argument *`<server>`* is the hostname of the machine running (or which will run) the RTC server the handle will represent.

`-program` *`<program>`* (default `0`)
Argument *`<program>`* is the program number on the given *`<server>`* the RTC server the handle will represent. Program numbers are discussed in Section 5.9.

`-transport` *`<transport>`* (default `udp`)
Argument *`<transport>`* specifies the network transport type the handle will use. Unlike RTC servers, clients can only use one transport type at once. The available transport types are system-dependent, but will generally be either `tcp` or `udp`, corresponding to the Internet protocols TCP/IP (connection-oriented) or UDP/IP (connectionless). Refer to Section 5.5 for a discussion of transport types. This option replaces the two former option flags `-tcp` and `-udp`.

`-timeout` *`<timeout>`* (default `25.0` seconds)
Argument *`<timeout>`* specified the maximum overall period of a remote call (executed using *`<handle>`* `call` below). If the server fails to respond within this period, a Tcl exception will be raised, and the call will return an error string. Refer to Section 5.6 for details on how to deal with the exception.

`-retry` *`<retry>`* (default `5.0` seconds)
Argument *`<retry>`* is only applicable to connectionless transports, such as UDP/IP, and is otherwise ignored. If a remote server fails to respond, the call is tried again after a delay of *`<retry>`* seconds. This process continues until the overall *`<timeout>`* period has elapsed. The value of *`<retry>`* is thus somewhat smaller then that of *`<timeout>`*.

`-recreate/-norecreate` (default `-norecreate`)
Setting `-recreate` mode forces the RTC handle to be 're-created' just before each remote call. This means the server port-mapper is re-consulted to locate the server port, and on connnection-oriented transports the old connection is closed and a new one opened. This is quite inefficient, but makes clients more robust in the face of server unreliability. It is particularly useful during server development.

`-debug/-nodebug` (default `-nodebug`)
Setting `-debug` mode causes miscellaneous debugging messages to be produced on the standard error stream, or the logging stream where there is one.

Note that only the syntax of arguments is checked: creating an RTC handle does not cause a remote call or any other network activity. Incorrect arguments (wrong server, program number, etc) will only be detected when the handle is actually used.

The handle created by `rtc`, `<handle>`, can be invoked with one of three action arguments, `call`, `configure` or `destroy`:

`<handle> call <name> [<arg> ...]`

Call the remote server function registered with the name `<name>`, passing it the given argument list, and returning the result string.

The RTC `<handle> call` function returns an error status (`TCL_ERROR`) if:

- the `<handle> call <name>` is called incorrectly (`<name>` is mandatory, although arguments for the remote function are not)
- the underlying RPC mechanism failure reports an error. There are several possible causes for this:
  - the remote server machine cannot be reached or does not exist
  - the server has no port-mapper daemon
  - the server isn't running the desired RTC server (identified by the RTC program number; see Section 5.9)
  - there is a communication failure, leading to loss of the request or response packet (more common on connectionless transports)
- the client's authorisation credentials are rejected by the server (see Section 5.11)
- the remote server does not have a function called *¡name¿*.

Note that these errors are related *only* to the RPC and RTC mechanism—which is concerned only with delivering requests to the user's remote server stub functions and getting the results back. Errors detected within the user's server stub functions *do not* cause Tcl errors to be returned. These 'user errors' can be signalled back to the calling client through the return string only, as they form part of the user protocol (see Section 5.1).

If an error occurred, the return string contains an error message indicating the source of error. Note that the call function may block for up to the `<timeout>` period before returning an error. Section 5.6 discusses a way of dealing with such errors in Tcl scripts.

`<handle> configure [-server <server>] [-program <program>]`
`[-transport <transport>] [-timeout <timeout>] [-retry <retry>]`
`[-recreate] [-norecreate] [-debug] [-nodebug]`

Alter the configuration of `<handle>` according to the given options, which are identical to those for the `rtc` function. Like the `rtc` handle creation function, configuring a handle does not cause a remote procudure call, or any other network activity. Only `<handle> call` invokes remote calls.

Calling `<handle> configure` with no arguments returns a string containing the current handle configuration.

`<handle> destroy`

Destroys the RTC handle, closing any connection with the server. The command `<handle>` is removed from the Tcl interpreter, so any subsequent attempt to use it will cause an error.

3.3 Client Side C Functions

The RTC system can also be used directly from a C program, using the functions described below. This defeats much of the purpose of RTC, but is necessary for situations where a computer does not have Tcl, yet should be a client of an RTC server.

The client side C functions correspond closely with the Tcl functions described above, because the Tcl loadable module is implemented using them. The RTC utility program `rtcclnt` (Section 4.2) allows these functions to be executed directly from a command-line environment.

In C programs, client handles are represented by objects of type `RTC`, which is a pointer to an opaque structure holding the configuration information and the RPC `CLIENT` pointer.

`RTC_STATUS rtcClientInit(void)`
: This function must be called before any of the other RTC client functions, because it initialises the RTC server data structures. It takes no parameters.

`RTC_STATUS rtcClientCreate(RTC *rtc, RTC_CONFIG tag, ...)`
: Creates a client handle object, and places a pointer to it in the location given by the `rtc` argument. The handle is then configured using the following arguments, a set of tag/argument pairs, terminated by a zero. The tags and argument types are:

Tag	Argument Type	Argument
`rtc_config_name`	`char*`	handle name
`rtc_config_server`	`char*`	server hostname
`rtc_config_program`	`int`	program number
`rtc_config_transport`	`char*`	transport type string
`rtc_config_timeout`	`int`	timeout in milliseconds
`rtc_config_retry`	`int`	retry in milliseconds
`rtc_config_recreate`	`RTC_BOOL`	recreate handle each call
`rtc_config_debug`	`RTC_BOOL`	produce debug messages

The meanings of the tags and arguments is exactly the same as the client options described in the previous sub-section in relation to Tcl client handles, except that here the timeout and retry periods are specified in milliseconds. The handle name argument relates to the Tcl *`<handle>`* name, and need not be used in C programs.

The same tag may appear more than once, per call; the last such tag/argument pair prevails. String data is duplicated by the function, and so need not be static. The `RTC_BOOL` flag has values `rtc_bool_true` or `rtc_bool_false` (1 or 0).

Errors in the configuration information, such as non-existant server hosts, wrong program numbers, and so on are generally only detected when a remote call is initiated by `rtcClientCall()` below.

`RTC_STATUS rtcClientConfigure(RTC rtc, RTC_CONFIG tag, ...)`
: Configure an extant client handle `rtc`. It accepts the same zero-terminated tag/argument list as `rtcClientCreate()`.

`RTC_STATUS rtcClientInterrogate(RTC rtc, RTC_CONFIG tag, ...)`
: Extract configuration information from an extant client handle `rtc`. It accepts the same tags as `rtcClientCreate()`, but with `pointers` to the appropriate user objects.

A pointer to a dynamically allocated duplicate string is returned into the user's pointer for string type configuration data. These strings should be subsequently deallocated by the user using `free()`.

`RTC_STATUS rtcClientCall(RTC rtc, int argc, char *argv[], char **resp)`
Perform a remote function call using the handle `rtc`. Pointers to the individual argument strings are in the array `argv[]`; `argv[0]` points to the name of the remote function to be called. Parameter `argc` is the number of arguments, counting `argv[0]`.

Parameter `resp` is a pointer to a string pointer, which after the call contains a pointer to a dynamically allocated string containing the result of the remote call, or if the call failed, a descriptive error message. A new response buffer is allocated for each call. The caller should subsequently free the buffers using `rtcClientResponseFree()` (*not* with `free()`).

If the remote server is not responding, this function will block for up to the currently specified timeout period, and will return an error status. Other errors may be detected immediately.

`RTC_STATUS rtcClientResponseFree(char **resp)`
Free the response buffer from a previous call to `rtcClientCall()`. Note that this function does not require the original client handle.

`RTC_STATUS rtcClientShow(RTC rtc)`
Print out the configuration information of handle `rtc`.

`RTC_STATUS rtcClientDestroy(RTC rtc)`
Destroy client handle `rtc`, deallocate its associated data structures and close any file/socket descriptors. The handle cannot be used again.

`RTC_STATUS rtcClientDone(void)`
This should be called when the user program has finished with the RTC library. Note that it does *not* close any outstanding client handles; use `rtcClientDestroy()` to do this first.

3.4 Other C Functions and Variables

Most RTC client and server C library functions return a status value of type `RTC_STATUS`. This has the following values:

Name	Value	Meaning
`rtc_status_ok`	0	function suceeded
`rtc_status_error`	-1	recoverable failure
`rtc_status_fatal`	-2	fatal error

The user should check the return status from each function call, and act appropriately. Most commonly, an error status is caused by a remote call timeout due to the server disappearing, or incorrect client configuration in the first place.

Fatal errors are caused by memory allocation failures, `NULL` pointers, corrupted data structures, and so on. The RTC library attempts to print a diagnostic trace to the standard error or logging stream. The user program generally cannot continue if this happens.

In either case, an error string can be obtained:

`char* rtcMessageGet(void)`

Return a pointer to a descriptive string indicating the cause of a recoverable failure or fatal error. This string should not be altered by the user.

4 Utility Programs

There are three utility programs for testing out RTC clients and servers. They run only on UNIX systems. These programs are run from the `bin` directory of the RTC distribution, once compiled.

4.1 Program `rtcsvc`

Program `rtcsvc` is an RTC server featuring only the two standard RTC remote calls, `ping` and `catalogue`:

```
rtcsvc [-p <program>] [-t <transport>]
    [-h <clienthost>] [-u <username>] [-g <groupname>] [-r <name>] [-d]
```

Starts an RTC server offering program number *<program>*, on the network transport type *<transport>*. Option `-t` may be repeated to add more than one transport type. The default (possibly system dependent) is to provide both TCP/IP and UDP/IP transports. The default program number is `0`.

The `-h` option adds host *<client>* to the server's host authorisation list. Option -u adds the UNIX user-id number for *<username>* to the user authorisation list, while -g adds the UNIX group-id number for *<groupname>* to the group authorisation list. These options may appear any number of times.

Option `-r` registers a stub function with the name *<name>*; this function behaves identically to the standard stub function `ping`, i.e. returns its own name. This option be repeated for different names.

Option `-d` enables the logging of debugging messages to the standard error or logging stream. The messages indicate the source of remote call requests, whether they passed the authorisation test, and if so, their arguments and reponse.

This program can be used for testing clients (to an extent) by acting as a placeholder for a projected server. In this case the `-d` option is used to add all the remote function names the server will provide: of course, the server's responses cannot be simulated.

4.2 Program `rtcclnt`

Program `rtcclnt` is an interactive shell-like program used to test servers, before the matching clients are completed. Its use was demonstrated in Section 2.4.

The command prompt displays the current RTC server hostname and program number (`rtcclnt` can only deal with one RTC server at once). The commands `server`, `program`, `transport`, `timeout`, `retry`, `recreate`, and `debug` are used to alter the configuration of the client, one item at a time. The client configuration can be printed out using `show`.

Command `call` initiates a remote call. The call's result string, or an error message, is printed out.

Program `rtcclnt` can execute canned sequences of commands, using the `script` command, or can execute any UNIX command using `shell`. In scripts, lines beginning with `#` are ignored, but there are no control structures (if you want them, use Tcl!).

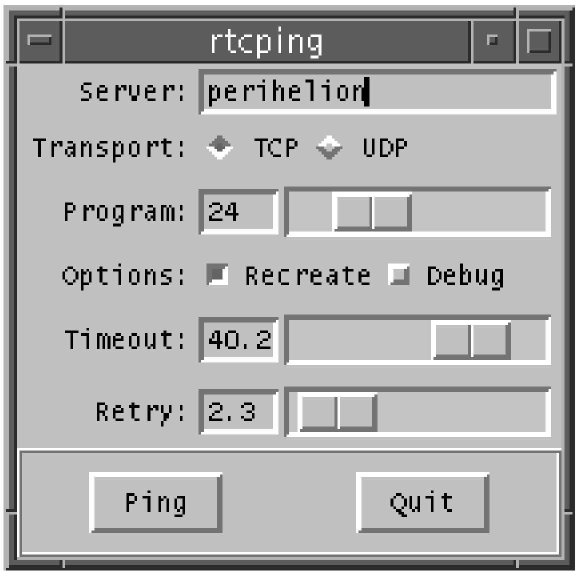
The command `help` print a complete list of commands and the arguments they take:

```
    % rtcclnt
    This is an RTC process
```

```
Library timestamp 970307125521
magni/0> he
Commands:
    call <name> [<arg> ...]        do remote call of <name> with <args>
    server <server>                set RTC server host to <server>
    timeout <period>               set timeout period (seconds)
    retry <period>                 set retry period (seconds)
    recreate [y|n]                 toggle/set recreate flag
    transport [tcp|udp|...]        set network transport type
    program <program>              set program number to <program>
    debug [y|n]                    toggle/set debug flag
    show                           print RTC client information
    source <file>                  read in/run RTC commands from <file>
    shell <command> [<arg> ...]    run UNIX <command> with <args>
    help                           print this message
    #                              lines beginning with # are ignored
    quit                           quit program
Commands can be abbreviated
magni/0>
```

4.3 Program `rtcping`

Program `rtcping` is a Tcl/Tk program, whose sole purpose is to test the `ping` remote call, but with a graphical interface, which looks like this:



Server: perihelion
Transport: TCP UDP
Program: 24
Options: Recreate Debug
Timeout: 40.2
Retry: 2.3

Operation is largely self-explanitary: the configuration of an RTC handle can be altered using the widgets at the top of the interface: a `ping` remote call is initiated by pressing the Ping button. If there is an error, a pop-up appears with the error string in it.

5 Practical User Interface Design using RTC

5.1 Designing the User Protocol

In traditional RPC programming, the user's protocol for a particular application is statically defined in a file with a formal protocol description language. Usually this file is then parsed by a program (such as `rpcgen`) which produces skeleton code for the client and server and for data encapsulation. The user then fleshes this out by writing server and client stub functions, gluing the RPC data transfer structures produced by `rpcgen` with the data structures of the application program.

With RTC the situation is much freer; there are really no data structures (although there may be structured data), and so the protocol is effectively defined by the set of stub functions the server provides, and the set or responses they will produce. This protocol can be thought of as the 'user protocol', as it is overlaid on the RTC RPC protocol, but defined by the user in much the same way as ordinary procedures and functions. There is still glue code to be written, but there is less of it, and it is easily debugged, because the user protocol is human-readable.

Despite there being no need for a formal protocol definition, it is worth planning and documenting it, like any other programming exercise. Unlike most RPC protocols, the user protocol can be extended dynamically—the server can install new stub functions at the request of clients, and clients can interrogate servers about their capabilities and adapt accordingly.

5.2 Server Stub Functions

The server stub functions receive their arguments as strings in the traditional `argc/argv` format. `argv[0]` is the name the function was invoked (registered) as; `argc` is always greater than zero; and `argv[argc]` is always `NULL`. Traditional string handling functions, such as `atoi()`, `atof()`, `strtol()`, `strtod()`, `strcmp()` and `sscanf()` can be used to analyse the arguments.

The server stub function's output is a single string, which can be constructed using `sprintf()`, `strcat()`, and so on. The function can return a constant string, a pointer into a static string buffer, a pointer into an argument string, or memory obtained using `rtcServerBuffer()` (Section 3.1).

Because there can only be one RTC server thread per address space (see Section 5.10) servers stubs can safely use local static buffers and return pointers into them. However, the single-thread restricion may one day be lifted, so this practice is not recommended.

Users can adopt any style of stub function argument passing they desire. While UNIX command-line options flags and arguments can be used (along with `getopt()`, although it is not reentrant), a positional style, where an argument's position in the argument list indicates what it is, is generally simpler and more effective.

Thus, `argv[1]` means one thing, `argv[2]` another, and so on. If arguments are optional, the appropriate argument can be passed a `DEFAULT` string, which the C stub functions chan check for and substitute the appropriate value. A more flexible arrangement is to arrange the arguments in order of decreasing usefulness, so that commonly specified argument come first and less commonly used ones (i.e., the more optional ones) come later.

Stub functions should be relaively relaxed about argument parsing, and should try and do something sensible with whatever arguments are passed, rather than returning error messages to the client. This minimises the need for error handling code on the Tcl side, which tends to clutter up scripts. In general it helps for the stub functions to be arranged to suit easy Tcl implementation, rather than the other way round.

5.3 Stateless Clients

In any network client/server arrangement, it is much easier if the clients can be made stateless, or at least mostly stateless. 'Stateless' means that they do not make any assumptions about the server's state, or try to maintain duplicates of it.

This is simply because the server state might change, but the client will not know about it. Servers might change their state unilaterally, or in response to other clients. As well, the client may lose communications with the server temporarily. The client cannot assume an element of server state is going to have the same as what it set it to last time. If it needs to use a data element, it must fetch it from the server each time.

Consider the situation of a server providing a remotely controlled voltage output, such as a digital-to-analogue convertor. In providing a remote graphical interface using RTC and Tcl/Tk, the obvious approach is to use a Tk `scale` (slider) widget controlling a Tcl variable. The `trace variable` function can then be used to arrange for a user function to be called whenever the variable changes, which does the remote call to update the server voltage. The value displayed on the slider should thus reflect the voltage coming out the digital-to-analogue convertor on the server.

This will be true, until another client changes the voltage. The second client will then be displaying the correct voltage, and the first will be wrong. The only way the first will be able to display the correct voltage is by querying the server to 'read back' the output voltage, as there is no way the server can tell the client it has changed. The only way the clients will *always* display the correct voltage is by continually querying the server, or in other words, polling.

Polling is not particularly elegant, but it is robust, and ensures that the client needs only very short-term display state. Section 5.8 shows how such polling can be put in the background.

On the other hand, there will be cases where a server response *can* be assumed to be fixed, and retaining the reponse as client state justifiable. For instance, in the example of Section 2, the list of serial port devices was assumed to be fixed, and fetched from the server only once. Since the number and names of device nodes on UNIX systems are quite static, the assumption is reasonable in this case.

5.4 Stateless Servers

Servers should also maintain the minimum of state required to do their job. In particular, they should not assume any particular behaviour from clients, such as expected sequences of remote calls, because such expectations will be confounded by two clients using the server at once. The server stub functions are not passed any information about the identity of the clients.

Servers should help their clients to be stateless. The remote functions they provide should make it convenient and efficient for clients to poll for displays of the server state. For instance, on the server controlling the digital-to-analogue convertor, there should be a function that *sets* the output to a given voltage, and another that *returns* the current voltage.

There should not be a function that, for example, *increments* the output voltage be a given amount, as to use this the client needs to assume it knows the current voltage, which might have changed since it last read it.

5.5 Connectionless versus Connection-oriented Transports

Allied to statelessness is the choice of network tranport protocols. Generally this is a choice between a connectionless (datagram) transport, such as UPD/IP, or connection-oriented (stream) transport, such as TCP/IP.

In general, connectionless transports are preferable for user interfaces, particularly during development. They ensure a 'loose' coupling between servers and clients, which can be killed and restarted without having to wait for connections to time-out and close.

However, they bring into relief the issues of statelessness discussed above. An RPC exchange on a connectionless transport is the exchange of two datagrams, the *request* and the *reply*. Either datagram might be lost in transmission. If a client has not received the reply for its request within the retry period, it sends the request again, and continues to send it until it gets a reply, or the overall time-out period has elapsed. Now, if it was the reply datagrams that were being lost in transmission, the server would have recieved a whole sequence of identical requests, which it would have faithfully executed. As long as one reply datagram makes it back to the server, the RPC exchange is regarded as sucessful.

In the case of our server with the voltage output, there would be no problem with the multiple requests if they were to simply *set* the voltage. The voltage would be set to the same value a number of times, but this would not matter. If the requests were to *increment* the output, the situation would be very different!

Connection-oriented transports bypass this problem (but this should be no excuse for writing stateful clients). They tend to give better performance on lossy networks, such as serial links and packet radio systems. However, on local ethernet networks they seem to have little advantage.

When using connection-oriented transports (or at least, TCP/IP), case is required to properly shut down the connection, otherwise one end (usually the client) will be left 'dangling', inoperative until the connection times-out, which can take some minutes. RPC servers generally don't clean up on exit very well, and on *vxWorks* and similar systems, the server is usually terminated by the reset and reboot of the whole computer. For these reasons, connectionless protocols are usually preferble.

5.6 Using `catch` around RTC calls

Tcl functions, including the RTC handle operations, return both a string result and a status code. The status code indicates if the procedure suceeded, or if there was an error; in the latter case, the result string is supposed to indicate what the error was. Errors are returned for conditions where the Tcl script cannot normally continue, such as when functions are given the wrong number of arguments, underlying system calls fail, and so on.

When a function returns an error, the Tcl interpreter halts execution of the script and prints out the return string as the error message, along with a stack dump. (Tk does much the same thing, but in a pop-up window). In most cases errors occur infrequently enough to leave their handling to the default Tcl mechanism. However, remote procedure calls require more careful treatment. Because they depend on network infrastructure, remote daemons and servers, and authentication processes working together properly, they are more likely to fail.

In a remote user interface application, handling errors of this nature is very important. The interface can be the user's only 'view' of a system, and problems and frustrations that occur here reflect badly on the system as a whole. The interface should never 'lock up', nor should the end user ever be presented with a stack dump.

The Tcl `catch` function can be used to deal with errors from RTC calls before the Tcl mechanism does. Assuming we have an RTC handle called `remote` and a remote function called `ls`:

```
    if [catch {remote call ls $args} result]
        # an rpc error occurred
        message\_popup "Can't contact remote server: $result"
        ...
    } else {
        # check for a user error
        if {[lindex $result 0] == "ERROR"} {
            message\_popup "There was a remote server error: $result"
            ...
        }
        # deal with a good response
        ...
    }
```

In this case, a message pop-up is thrown up if an error is caught, which is only a marginal improvement on the default Tk mechanism. However, the pop-up could, for instance, permit the user to select a different server and retry the operation.

Remember that RTC calls return Tcl errors only when the RTC or RPC mechanism fails, and not when the user server stub functions report errors through their return string. In the example, the first token of the return string is checked for the word `ERROR`, but the user may choose to employ any protocol (Section 5.1).

5.7 Using the server `catalogue`

All RTC servers provide a `catalogue` function, which returns a list of the names of all functions registered on the server. This can be used by clients in various ways.

The `catalogue` functions can be used as part of the user protocol, in cases where servers offer a variable number or range of services which must be adapted to by a single client.

The example given in Section 2 is like this: the client doesn't initially know how many terminal devices the server has, or what their names are. An alternative implementation could have the server register an RTC stub function for each terminal device available, rather than `listports` and `message`. The client would use `catalogue` to get the available functions (effectively, the list of ports), and would call one of the functions directly to print the message.

If users do not like the `<handle> call <name>` call syntax, they may insert <name¿ into the Tcl interpreter directly, by defining a procedure:

```
    proc name args {
        if [catch {remote call name $args} result] {
            ...
        }
        ...
        return $result
    }
```

This has the advantage of hiding the `catch` mechanism (and user error handling as well, if desired). The `catalogue` function can be used to automate this process, inserting *every* remote function offered by the server into the interpreter:

```
if [catch {remote call catalogue} allfuncs] {
    ...
}
foreach func $allfuncs {
    proc $func args {
        if [catch {remote call $func $args} result] {
     ...
        }
        ...
        return $result
    }
}
```

In this case, the remote functions have to be fairly consistent in their operation, at least with respect to errors. It is also important that the names the stubs are registered with in the server do not clash with any extant functions or variables on the Tcl client side, or contain illegal characters.

5.8 Background Polling for Remote Information

Often a Tcl/Tk interface to a system must maintain an up-to-date display of the remote systems state or process variables. Unfortunately, the server cannot tell the client when this state has changed and needs to be re-displayed. The only alternative is for the client to poll the server at regular intervals, and re-display the data if it has changed.

This process can be 'backgrounded' using the Tcl `after` function. After a script has initialised everything, it calls a function `poll`:

```
proc poll {{period 1000}} {
    display_update
    after $period poll $period
}
```

Function `display_update` uses RTC to get new data from the server and display it. Then `poll` scedules itself to be *called back* by the Tcl interpreter after `period` milliseconds, and so on.

To be workable, this scheme needs a little refinement. Polling needs to be disabled while Tk widgets (such as scales) are being adjusted, or strange things happen. This can be done by `binding` a function to the button press/pointer leave events of the affected interactors.

The PIRAT user interface Tcl scripts (Section 5.12) used polling for display update. A generic polling module `poll.tcl` was used (see the sources).

5.9 Program Numbers and the RPC Port-Mapper

The RTC program numbers, which are assigned by the user, are equivalent to RPC service numbers, which differentiate one RPC service from another. The RPC service numbers used by RTC servers and clients are simply the RTC program numbers added to 600 000 000 decimal.

This allows RTC to make use of the standard RPC port-mapper daemon that is present on all systems equipped with RPC. (This daemon is variously called `portmap`, `rpc.portmapd` or `rpcbind`). The port-mapper is the clearing-house between RPC clients and servers, and is itself an RPC server.

It operates as follows: a server program initially registers itself with the port-mapper on its machine, sending it the RPC service number for the protocol it proposes to serve. The port-mapper returns a new Internet port number, and the server establishes itself, listening for requests on that port. Clients seeking a particular RPC service initially contact the port-mapper on the server machine, sending the number of the RPC service they desire. The port-mapper returns the port number the server should be using, which the client then uses to complete the user's remote call.

Clients usually only need to deal with the port-mapper once in their lifetime, because server processes usually have longer lifetimes than clients. During server development, however, the reverse can be true. The RTC 'client-recreate' option (in its various forms) can be valuable in this situation.

The port-mapper only remembers a single port number for each service number. If a second server of the same service registers itself, the previous registration for that service is lost. However, clients can continue to use the first server on the original port, because the new server will be given a different port number. New clients will be given the new server's port number.

The port-mapper also does not trace server processes. If a server exits, it should in principle contact the port-mapper daemon to un-register itself, or the daemon will continue to report the abandoned port number to potential clients. Unfortunately, an exiting server can inadvertantly un-regisiter a newer server of the same service number, resulting in no such servers being registered.

In general, more than one RPC/RTC server of a given service/program number per machine should be avoided.

5.10 Multi-threaded Clients and Servers

The RTC client library is itself fully re-entrant, and may be safely used in multi-threaded C programs, assuming the underlying RPC library can be. This varies from platform to platform, and may apply for some operations and not others.

The RTC server is not fully re-entrant, and only one thread per address space may enter the server loop. ('Address space' means a process on a UNIX-like system, or a whole system for a *vxWorks*-like shared-memory kernel). This is mainly because the RPC server loop on most platforms is not re-entrant.

This does not mean that RTC server processes cannot be multi-threaded, just that there can be only one thread being the server. Other threads can call RTC server functions, such as `rtcServerRegister()` and `rtcServerAuth()`, at any time, as the server lists are properly protected by a monitor semaphore.

Server stub functions are called in the context of the server thread. It is up to the programmer to use monitors in the stubs if they access thread-shared data, which is fairly likely. Remember, however, that stubs should not block (at least not for long), because the whole server is affected.

The RPC implementation on Solaris can permit the server to spawn a new thread for each request; this mode is intended for high throughput RPC servers with client functions that may block. It not presently supported by RTC.

5.11 Server Access Control

RTC offers a more sophisticated authorisation control scheme than is generally used in RPC servers, which by default respond to requests from 'all comers'. The intention is to limit server reponse only to requests from clients on certain machines and having certain UNIX user-id and group-ids. By default, only requests eminating from the same machine and with the same user-id and default group-ids as the server has will be honoured (and then, only if the service number described above matches). If the range of acceptable machines, users or groups is to be extended, as is usually the case, the the RTC authorisation functions described in Section 3.1 need to be used.

The server maintains three acceptance lists, one for machine host-names, one for user-ids, and one for group-ids. There is a function to add an element to each list. When a client sends an RTC request, the RPC mechanism sends with it an *auth_unix* credential. This structure contains the client machine name, the user-id of the sender, and the group-ids of all the groups the sender is a member of. The server tests the credential to ensure

- the client's host-name is in the server's host-name list, AND
    - the client's user-id is in the server's user-id list, OR
    - the client's effective group-id is in the server's group-id list, OR
    - any of the client's group-ids are in the server's group-id list.

If the test fails, the server refuses the request, and returns a code to the client indicating that its credentials were not acceptable (`AUTH_BADCRED`).

This authorisation system cannot be regarded as 'secure', since *auth_unix* credentials are easily forged. (The more secure *auth_des* system is unfortunately not widely available). Its purpose is not to increase security from malicious attack—although it is an improvement over normal RPC practice—but to improve security against accidental errors by developers, or where two users using the same RTC application (or service number) on different machines.

This flows from the observation that RPC protocols have occasionally been used to control 'real' equipment, such as robot arms, so the security of access to the server is an obvious factor in the safety of such systems. Equipment of this kind **should never be controlled through a computer network** unless adequate fail-safe local safety provisions are in place, such as power-interlocked access gates and emergency-stop buttons.

5.12 RTC in the pirat System

The RTC distribution includes the Tcl/Tk user interface scripts used by the PIRAT sewer inspection system. These are of no particular use without the corresponding server, but are useful to illustrate an interface to a complicated system built using an early version of RTC.

There are four separate programs dealing with the four main operational areas of the PIRAT system: power and data gathering (master mode), vehicle and winch (operator mode), laser scanner, and sonar scanner. These four separate interfaces ran on a Sun workstation, and communicated with an RTC server running as a task on a VMEbus *vxWorks* real-time system. The server provided over twenty remote functions controlling or monitoring individual aspects of the disparate system. The interface programs collected these functions together into a unified presentation, allowing semi-skilled (or sleep-deprived) operators to control the system in relative safety.

6 Conclusion

The remote graphical user interface has become something of a standard feature of systems constructed within the Division's Industrial Automation Programme. While the appropriateness of such interfaces can occasionally be questioned, there is little doubt their provision consumes significant programmer time. The appearance of Tcl (Tool Control Language) and its associated X Windows toolkit Tk prompted the author to review the mechanisms which these interfaces are implemented. The result was the Tcl extension library RTC (Remote Tool Control), which has been the subject of this report.

RTC is basically a fixed RPC protocol with a client interface integrated with Tcl, and a pre-written, run-time extendible server core. Server functions are written in C to a standard prototype and registered with the server. These functions can then be remotely called from a Tcl script almost as simply as a normal Tcl function.

The client side of RTC can be used on any system that provides the Tcl interpreter and the standard RPC client libraries—almost any UNIX system. Clients can also be written in C, avoiding the use of Tcl. Implementation on Microsoft Windows platforms may be feasible. The server libraries can be compiled for most Unix systems that provide RPC, and will also run on small shared-memory multi-tasking kernels, such as *vxWorks*, also provided that they have RPC.

RTC was used to construct the main operator interfaces for the PIRAT Instrument System, which provided an ideal vehicle to test the initial implementation and develop effective ways of using it. The PIRAT interfaces were developed surprisingly quickly, vindicating the RTC approach.

References

[1] G. Campbell, K. J. Rogers, and J. Gibert. PIRAT Project, Quantitative Sewer Inspection—Stage 2: Development and Assessment of a System for Field Use. Confidential Technical Report MTM-415, CSIRO Division of Manufacturing Technology, Melbourne, May 1995.

[2] On-Line Applications Research Corporation. *Real Time Executive for Military Systems: C Applications User Guide.* U.S. Army Missile Command, Redstone Arsenal, AL 35898-5254, July 1994.

[3] SunSoft Inc. *OpenWindows Version 3: XView Reference Manual.* Sun Microsystems Inc., 2550 Garcia Avenue, CA 94043, U.S.A, 1991.

[4] SunSoft Inc. `rpcgen` Programming Guide. In *SunOS Network Programming Guide.* Sun Microsystems Inc., 2550 Garcia Avenue, CA 94043, U.S.A, 1991.

[5] R. J. Kirkham and P. I. Corke. Buderim Ginger Project Technical Report: ROBOSORTER Overview. Confidential Technical Report MTM-371, CSIRO Division of Manufacturing Technology, Melbourne, December 1994.

[6] R. J. Kirkham, P. I. Corke, and H. Nguyen. Buderim Ginger Project Technical Report: The ROBOSORTER Machine Vision System. Confidential Technical Report MTM-425, CSIRO Division of Manufacturing Technology, Melbourne, September 1995.

[7] R. J. Kirkham and A. J. Dreier. Data Handling and Control Software. PIRAT Project Stage 2 Technical Report MTM-404, CSIRO Division of Manufacturing Technology, Melbourne, May 1995.

[8] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[9] Wind River Systems. *vxWorks Programmer's Guide.* Wind River Systems, Inc, 1010 Atlantic Avenue, Alameda CA 94501-1147, 1992.