Chapter 7
Interacting With Other Programs

So far, we have talked about writing individual programs. In this chapter, we'll show how Tcl/Tk can interact with other programs. External applications can be called to perform tasks, and applications can be split into cooperating front-end and back-end processes. These processes can run on the same computer or across a network.

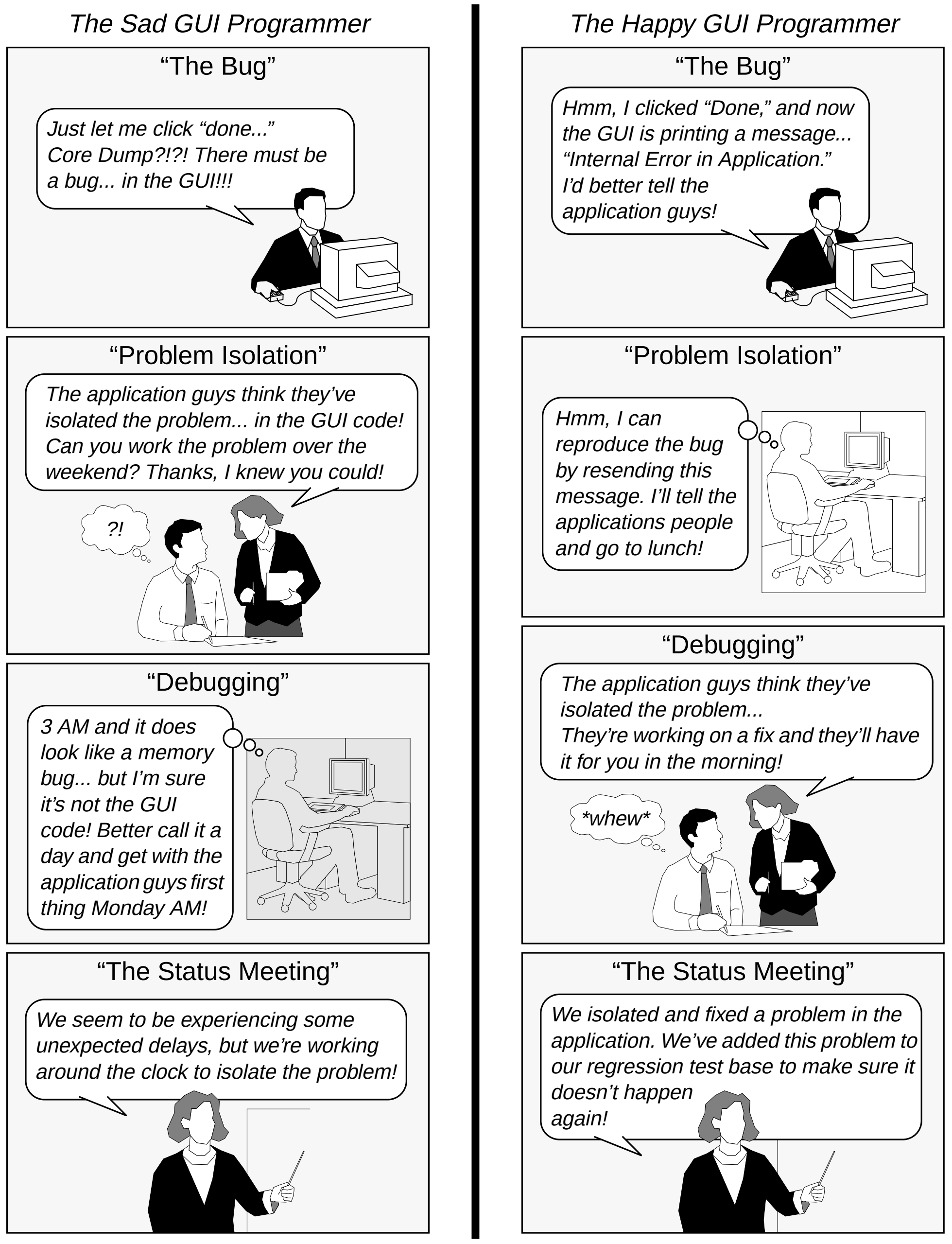
If you're wondering why all of this is important, consider the *Tale of Two GUI Programmers*, shown on the following page. The Sad GUI Programmer mixes his GUI code in with the rest of the application. When someone finds a bug in the program, he gets blamed for it. If the program crashes, the GUI is an immediate suspect. The Happy GUI Programmer, on the other hand, uses the ideas described in this chapter. He has a front-end GUI process controlling a separate back-end application. When someone finds a bug in the application program, he tells the back-end guys to fix it, and goes to lunch.

7.1 Executing Other Programs

Let's start by looking at how Tcl/Tk can be used to drive other programs. Suppose your application has an extensive on-line help system. You might want to minimize the storage space needed for all of the help files by installing them in compressed form. On Unix systems, you can compress the help files using the `compress` program, and you can display a compressed file using the `zcat` program, which prints the original file to its standard output.

Now suppose we want to display one of these help files in a Tcl/Tk application. We can use the `textdisplay` procedures developed in Chapter 5 to display the help text, but how do we decompress it and read it in? We can use the `exec` command to execute the

A Tale of Two GUI Programmers


The Sad GUI Programmer
"The Bug"
Just let me click "done..." Core Dump?!?! There must be a bug... in the GUI!!!
"Problem Isolation"
The application guys think they've isolated the problem... in the GUI code! Can you work the problem over the weekend? Thanks, I knew you could!
?!
"Debugging"
3 AM and it does look like a memory bug... but I'm sure it's not the GUI code! Better call it a day and get with the application guys first thing Monday AM!
"The Status Meeting"
We seem to be experiencing some unexpected delays, but we're working around the clock to isolate the problem!
The Happy GUI Programmer
"The Bug"
Hmm, I clicked "Done," and now the GUI is printing a message... "Internal Error in Application." I'd better tell the application guys!
"Problem Isolation"
Hmm, I can reproduce the bug by resending this message. I'll tell the applications people and go to lunch!
"Debugging"
The application guys think they've isolated the problem... They're working on a fix and they'll have it for you in the morning!
whew
"The Status Meeting"
We isolated and fixed a problem in the application. We've added this problem to our regression test base to make sure it doesn't happen again!

A Tale of Two GUI Programmers: Epilogue

❑ While the application guys were fixing their bugs, the Happy GUI Programmer wrote several profitable applications and was generously rewarded by his employer.

❑ The Happy GUI Programmer's manager won several prestigious quality awards and went on to lead a large division of the company.

❑ The Sad GUI Programmer learned to write two-process GUIs and eventually retired happily to a small cottage in the country.

❑ The application guys are still trying to figure out what's wrong.

`zcat` program from within our Tcl/Tk application. The `exec` command executes another program, and returns its standard output as a Tcl result string. If the program fails (i.e., the program exits with a non-zero status code), the `exec` command returns an error.

Implementing an on-line help system with compressed files boils down to just a few lines of code:

```
set help [textdisplay_create "On-line Help"]

set info [exec zcat overview.Z]
textdisplay_append $help $info
```

First, we use the `textdisplay_create` procedure to create a text display dialog with the title "On-line Help." This procedure returns the name of the new dialog. We'll need this name when we want to refer to the dialog, so we store it in a variable called `help`. Next, we use the `exec` command to execute the `zcat` program for a help file called *overview.Z.* This returns the entire contents of that file, which we store in a variable called `info`.[1] Finally, we call the `textdisplay_append` procedure to load the information into the help dialog.

7.1.1 Execution Pipelines

Many Unix programs are designed to work together. The output from one program can be fed to the input of another. For example, suppose that our help files are not just plain text, but instead they are in the `nroff` format used by Unix manual pages. On top of that, they are compressed. We can use the `zcat` program to decompress them, and feed the output

1. This may be few lines, or several mega-bytes of information. Tcl won't complain unless your computer runs out of memory to store the information.

to the `nroff` program, which formats the text. The `nroff` program inserts some control characters to handle bold and italic text, but we can strip these out by feeding its output to the `colcrt` program. The result of all this will be a very long string of help information that we can display in our on-line help window.

The `exec` command supports the execution of several programs in a pipeline. Programs are separated in the `exec` command by the pipe "`|`" symbol, indicating that the output of one program feeds to the input of the next. We could modify our help system to display compressed, nroff-format files like this:

```
set help [textdisplay_create "On-line Help"]

set info [exec zcat eccsman.Z | nroff -man | colcrt]
textdisplay_append $help $info "typewriter"
```

Again, we use the `textdisplay_create` procedure to create a help window. This time, instead of executing the `zcat` program by itself, we put it in a pipeline along with `nroff` and `colcrt`. All three of these programs operate on the help file, producing the result that we store in the `info` variable. Finally, we call `textdisplay_append` to display the help text. We add the `"typewriter"` argument so that the text will be displayed in a constant-width font, as is customary for manual pages.

7.1.2 Building Commands and Handling Errors

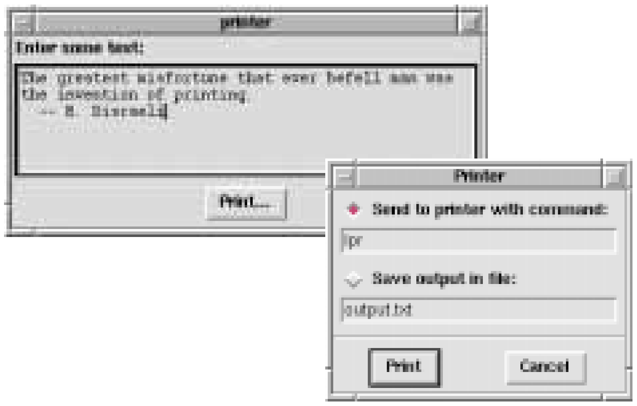
Now that you understand how the exec command works, let's use it in a more complicated example. Suppose we have an application which has a print function, as in Figure 7.1. We've already explained how the printer dialog box is constructed in Section 6.6.3, so we'll skip the details here, and concentrate on the mechanism that is used to do the actual printing.

We can execute a Unix program like `lpr` to handle the printing. For example, we can use the following command to print a simple message:

```
exec lpr << "The greatest misforture that ever befell man was
the invention of printing.
-- B. Disraeli"
```

The `<<` characters redirect the string to the standard input of the `lpr` program, which then routes the string to a printer.

But for the program shown in Figure 7.1, we don't want to hard-code the `lpr` program or the message. Instead, we'll handle things in a generic way. As you may recall from Section 6.6.3, we set up the `printer_print` procedure to take a command as one of its arguments. When you press the *Print* button, the `printer_print` procedure executes this command to get some text and then routes the text to the printer. In this example, the command will return the text from the text widget in the main window. But you can pass any command into the `printer_print` procedure, so you can use this dialog to print things from any application. And you can change the printer program in the dialog, so you can route your output to many different printers.



```
The greatest misfortune that ever befell man was
the invention of printing.
  -- B. Disraeli
```

Send to printer with command: lpr

Save output in file: output.txt

Figure 7.1. Printing a bit of text. We `exec` the `lpr` program to send the output to the printer.

In order to do all of this, we need to execute a command within `printer_print` that looks like this:

`exec` *program* `<< [`*command*`]`

Here, *program* is the printing program from the entry widget on the printer dialog, and *command* is the command that you pass into `printer_print`. For a particular printing task, We can substitute in the *program* and the *command*, and build a print command that looks just like the one above. Our `printer_print` procedure uses the following code to build a print command:

```
set print "exec [$info.printerCmd get] << \[$cmd\]"
```

If you look back at the `printer_create` procedure in Section 6.6.3, you'll see that the entry widget containing the print program is `$info.printerCmd`. So we use the command `[$info.printerCmd get]` to get the contents of this widget. Also, the `cmd` variable contains the command passed into `printer_print`. We substitute these things into the `exec` command string, building a command and storing it in the `print` variable. We'll use this command in a moment to do the actual printing.

Notice that we escaped the square brackets around `$cmd` with a backslash, so they will appear in the final `exec` command, as we showed earlier. That way, we will execute the *command* string and direct its output into the *program* for `exec`. Whenever you build up a code fragment like this, you run into the following problem: There are some things that you want to substitute when you build the command, and others that you want to substitute later, when you execute the command. You can put backslashes in to handle this, as we did in the example above, but this can be quite confusing.

There is another way to build up the print command, without the backslashes. We can use the `format` command to substitute parameters into the command string, like this:

```
set print [format {
    exec %s << [%s]
} [$info.printerCmd get] $cmd]
```

The first argument to the format command is a template for the string that we're building. Within this template, each `%s` marks the spot where a parameter will be substituted. The first `%s` will be replaced with the result from `[$info.printerCmd get]`, and the second with the value from `$cmd`.

Once we've built up the print command, we can execute it to handle the actual printing. If anything goes wrong in the print program, the `exec` command will return an error. We can use the `catch` command to execute our print command and catch any errors that might result. So the code within `printer_print` that actually does the printing looks like this:

```
if {[catch $print result] != 0} {
    notice_show $result error
} elseif {[string trim $result] != ""} {
    notice_show $result info
} else {
    notice_show "Document printed" info
}
```

The `catch` command executes our print command `$print` and returns its status code. If the status code is non-zero, then something went wrong. For example, the `lpr` program could return an "unknown printer" error. In that case, the `result` variable contains the error message "unknown printer," and we use the `notice_show` procedure developed in Section 6.4 to display that message with an "error" bitmap.

Even if the print program executes successfully, it could return some useful information. For example, the `lpr` program might return a message like "printer request is lp-210". In that case, the status code will be zero, but again, the `result` variable will contain the message. Again, we pop up a notice dialog to display the message with an "info" bitmap. Of course, even if the program doesn't issue any diagnostic message, we'll display a message like "Document printed" indicating that the job has been printed.

Note that in addition to using `exec` as we have here, we can use the `open` command to execute other programs. We'll cover that approach in the sections that follow.

7.2 Collecting Output from Long-Running Programs

Using `exec` is fine for short-lived programs that produce a small amount of output, but for long-running programs, it suffers from two problems:

- Your program will block, waiting for the `exec`'d program to finish before resuming its execution.

- The output of the program will be returned to you in one chunk, when the `exec`'d program has finished.

The first issue is very serious when writing a graphical user interface, since no event processing takes place while you are waiting for the subprocess to complete. In other words, your program won't be responsive to the mouse or the keyboard, and its windows won't refresh themselves properly when you cover and uncover them. This is both ugly and confusing to the user, who may conclude that your program has died.

The second problem is an issue with programs that run for a long time and produce periodic output. For example, the point-to-point protocol (PPP) has become a popular way of establishing an internet connection via dialup modems. On many Linux systems, the program `pppstats` is used to monitor the PPP connection and print statistics about the throughput of the dialup link. When it is running, it updates its status information every so often by printing a line of information, containing statistics on the number of packets sent and received, the number of errors, and the input and output throughput in bits per second. Its output looks like this:

```
$ pppstats
    in   pack   comp uncomp    err  |    out   pack   comp uncomp     ip
1397955   1500    726    577      0  | 231377   1518    506    777    235
 15792     19     15      4      0  |  12151     17     11      6      0
 14376     20     15      5      0  |  15118     20     11      9      0
 14966     18     12      6      0  |  12134     16     10      6      0
 14273     17     14      3      0  |  10630     17     12      5      0
 15824     19     14      5      0  |  12243     20     12      8      0
 ...
```

This program is a good candidate for a graphical front end. We can read its output, filter out everything but the `in` and `out` columns, and display these values in a line chart.

The `pppstats` program runs forever, so we can't just `exec` it and wait for it to finish. And even if `pppstats` did terminate, we would get its output all at once, instead of getting it a little at a time, as it becomes available.

To handle the output properly, we must use the `open` command in place of `exec`. Normally, the `open` command opens a file and returns its file handle. But it can also execute a process and return a file handle for its input/output. So you can read from or write to the process as if it were an ordinary file. To do this, you simply replace the usual file name with an execution pipeline that starts with the pipe "`|`" symbol, like so:

```
open "|pppstats" "r"
```

The usual `open` command options apply to processes as well as files, so that we can open the process either for reading (`"r"`), for writing (`"w"`), or for reading and writing (`"r+"`). In this case, we've opened the `pppstats` program for reading, so we can monitor its standard output.

Once we've started the subprocess, we need to know when it has output available. Remember, the `pppstats` program only prints a line of information every so often. We can use the `fileevent` command to know when the output has arrived. The `fileevent` command is just like the `bind` command in Tk, but it works with file handles. It registers a script with the event loop, and when a file handle becomes readable or writable, it executes the script to handle that event.

For example, we can use the following command to handle each line of output from `pppstats`:

```
fileevent $fid readable "get_samples $fid"
```

Whenever the file `$fid` becomes readable, the command `get_samples $fid` is executed to handle the data.

Now let's look at `get_samples`:

```
proc get_samples {fid} {
    if {[gets $fid line] >= 0} {
        if {[scan $line "%d %*d %*d %*d %*d | %d" in out] == 2} {
            pppmon_add $in $out
        } elseif {[lindex $line 0] == "in"} {
            gets $fid line    ;# skip cumulative totals
        }
    } else {
        close $fid
    }
}
```

It takes one parameter, which will be the file handle returned when we opened the `pppstats` program. Note that since the actual value of `$fid` is stored as part of the command, we can avoid having to access the file handle as some kind of global variable.

The first thing the procedure does is read a line of text and check for end-of-file. If the `gets` command returns a negative number, then the `pppstats` program has terminated. Detecting this isn't really necessary for our demo program (`pppstats` won't exit unless it has a bug of some sort), but it's good form, and it's required in cases where a subprocess is expected to exit at some point.

Once we have read the line of output, processing is simple. If the first word of the line is `"in"`, then we have received the header line and we just skip it. Otherwise, we have received a line of data. We use the `scan` command to extract the data from the first and sixth columns (the input and output data transfer rates) and we call the `pppmon_add` procedure to plot it. There's nothing special about using `scan`—we could just as easily have used `regexp`, `lindex`, or any of the other string manipulation commands to extract the two fields.

Sometimes you can use your knowledge of the invoked program to simplify your processing—especially if you wrote the invoked program! For example, if you knew for certain that `pppstats` only prints its header one time, you could add a couple of `gets` statements to skip over the header lines, like this:

```
set fid [open "| pppstats" "r"]
gets $fid ;# eat header line
gets $fid ;# eat cumulative total line
fileevent $fid readable "get_samples $fid"
```

If you wrote `pppstats`, or if you have the source code for it, you might even modify it to work this way. Unfortunately, `pppstats` was designed to be run from a terminal, so it repeats the header every so often, and we are forced to work around it.

Using some of the concepts we discussed in Chapter 4, it is a simple matter to write a plotting routine to draw a line chart on a canvas. We won't show the plotting code here, but it is contained in the `pppmon` script, in the source code that accompanies this book. When you run this program, it produces a nice PPP monitor, as shown in Figure 7.2.

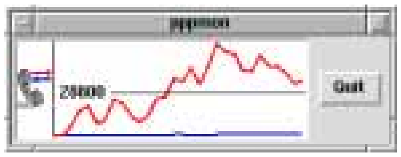


Figure 7.2. Monitoring PPP output by creating a pipe to `pppstats`.

7.3 Driving Other Programs Without Temporary Files

In the same way that we can open programs for reading, we can also open programs for writing. We simply open the pipe in "write" mode by changing the access parameter to `"w"` like this:

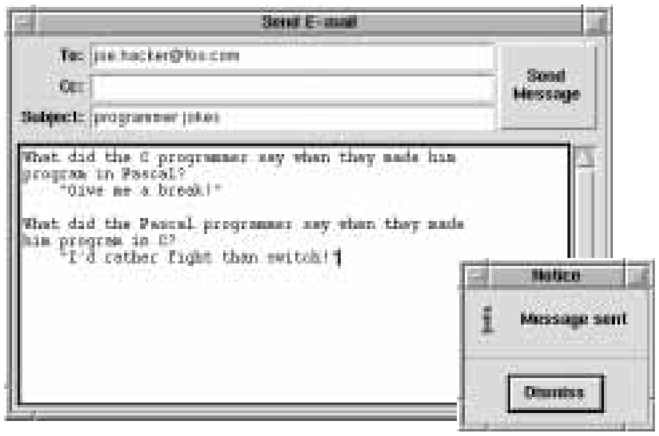
```
set fid [open "|program" "w"]
```

Anything written to the file handle `$fid` will be sent to the standard input of *program*.

As an example, suppose you're writing a program to send electronic mail (e-mail), like the one shown in Figure 7.3. With this application, you fill in an e-mail address, a subject line, and a message, and then you push the *Send Message* button to send out the mail message.

On Unix systems, you can use programs like `mail` and `sendmail` to send out e-mail. You might be tempted to write the mail message to a temporary file, and then execute `sendmail`, feeding it the file. But with the pipe facility in Tcl, things are much simpler. You can use the `open` command to execute the `sendmail` program, and then write the message directly to its standard input. For example, suppose we handle this with a procedure called `email_send`:

```
proc email_send {to from cc subject text} {
    set fid [open "| /usr/lib/sendmail -oi -t" "w"]
    puts $fid "To: $to"
```



To: joe.hacker@foo.com
Cc:
Subject: programmer jokes

```
What did the C programmer say when they made him
program in Pascal?
    "Give me a break!"

What did the Pascal programmer say when they made
him program in C?
    "I'd rather fight than switch!"
```

Message sent

Figure 7.3. Sending mail by creating a pipe to `sendmail`.

```
        if {[string length $from] > 0} {
            puts $fid "From: $from"
        }
        if {[string length $cc] > 0} {
            puts $fid "Cc: $cc"
        }
        puts $fid "Subject: $subject"
        puts $fid "Date: [clock format [clock seconds]]"
        puts $fid ""  ;# sendmail terminates header with blank line
        puts $fid $text
        close $fid
    }
```

This procedure takes five arguments: e-mail addresses for the recipient and the sender; a comma-separated list of e-mail addresses for anyone who should get a copy of the message; the subject line; and the body of the message. We simply open the `sendmail` program and write out the message header and its body. When we close the connection, `sendmail` sends off the mail message and terminates.

Notice that we didn't put any error checking in this routine. Since this is a fairly generic procedure, we didn't want to add any application-specific error reporting. That would make it difficult to reuse this procedure in other applications. So there is no point in catching the errors from the `open` or `close` commands. Instead, it is better to let the errors filter back up to the calling routine, and handle them there with a single `catch` command.[2]

For example, the e-mail application shown in Figure 7.3 uses the following bit of code to actually send the mail message:

```
...
set cmd {email_send $to $from $cc $subject $text}
if {[catch $cmd result] != 0} {
    notice_show "ERROR:\n$result" error
} else {
    notice_show "Message sent"
}
```

The `catch` command catches any error that occurs within the `email_send` procedure. If an error is encountered, we pop up a notice dialog to report it. Otherwise, we pop up a notice saying that the message was sent.

7.4 Working Around Buffering Problems

If you are reading output from a program, you may find that it comes in large chunks, instead of coming in a little at a time. Many programs keep their output in a buffer, and they only flush the buffer when it is full. This makes the programs much more efficient, but it causes many headaches when you want to monitor the output of the program in real time.

This problem is sometimes hard to diagnose. Many programs will refrain from buffering when their output is destined for the screen, since presumably a human will be watching. But the same programs will buffer their output if the destination is a file or another program, since in that case it's more efficient to conserve system resources. The C standard I/O library does this kind of smart buffering, so the majority of C programs have this behavior built in.

7.4.1 Seeing The Problem

Suppose the following C program is stored in a file called *bufdemo.c*:

```
#include <stdio.h>
main() {
    while (1) {
        printf("This is a line of text which might be buffered\n");
        sleep(1);
    }
}
```

2. This is one of the advantages of Tcl's exception-style error reporting— intermediate layers of software can safely defer error handling to calling routines.

On Unix systems, you can compile this program with a command like:

```
$ cc bufdemo.c -o bufdemo
```

This produces a program called `bufdemo` that we can use in the following experiment.

To see the effects of buffering, try running this program two different ways. First, run it from the command line without redirecting the output:

```
$ bufdemo
```

You will see that every second, the program prints one line of output. Now try running the program with this command line:

```
$ bufdemo | cat
```

Since the program's output is not going directly to the terminal[3], the output will be buffered. The `cat` program on the receiving end of the pipe will not get any input until `bufdemo` program's output buffer is filled. The actual size of the buffer will vary by system, but buffers of 4K to 8K are typical. Assuming an 4K buffer, the program will seem to be idle for about 1.5 minutes[4], spit out a bunch of text, sit idle for another $1^{1}/_{2}$ minutes, and so forth. If you wrote a graphical front end for the `bufdemo` program you would see the same behavior.

Now suppose that this same program were written in Tcl instead of C. It might look like this:

```
while {1} {
    puts "This is a line of text which might be buffered"
    after 1000
}
```

If this program were stored in a file called *bufdemo.tcl*, you could run it like this:

```
$ tclsh bufdemo.tcl
```

and it would write out one line each second, like the previous `bufdemo` program. Again, try piping its output to the `cat` program, like this:

```
$ tclsh bufdemo.tcl | cat
```

You might think that the output would arrive in chunks, as it did in our last experiment. But in this case, there doesn't seem to be a buffering problem. By default, Tcl flushes the standard output buffer when you write out each line. It does this whether the output is sent to a terminal or to another program.

But there is still a buffering problem lurking about. Suppose that we modify the program slightly to look like this:

```
set fid [open "| cat" "w"]
```

3. Many systems use the `isatty` system call to determine this.

4. 4096 character buffer size / 46 characters per line = 89 lines to fill up buffer. Since one line is printed each second, it takes 1.5 minutes to fill the buffer. Since we're using the `after` command to implement the delay, this number should be unaffected by differences in computer speed.

```
while {1} {
    puts $fid "This is a line of text which might be buffered"
    after 1000
}
```

This time, instead of writing to standard output, we open a pipe to the `cat` program and write directly to it. In this case, the output to the `cat` program is fully buffered, so again, the output will appear in chunks every 1.5 minutes.

7.4.2 Fixing the Problem

Now that you've diagnosed the problem, what can you do about it? Take a look at the program that is producing the output.

- If the program is under your control, and if it is written in C, C++ or Tcl, just make sure that you flush the output after writing each line.

  In C, you should call the `fflush` procedure (part of the standard I/O library) after printing each line. In C++, you can simply terminate each line with the `endl` manipulator for the `ostream` class, and it will flush automatically. In Tcl, you can either call the `flush` command after each `puts`, like this:

  ```
  set fid [open "| cat" "w"]
  puts $fid "This line of text is immediately flushed"
  flush $fid
  ```

  Or better yet, you can use the `fconfigure` command to change the buffering mode for a file handle, like this:

  ```
  set fid [open "| cat" "w"]
  fconfigure $fid -buffering line
  puts $fid "This line of text is immediately flushed"
  puts $fid "So is this line"
  ```

  In this example, we set the buffering mode to `line`, so each line of output is flushed automatically.

- If you are using a program written by someone else, or if for some reason you can't flush the output (this may be a problem with some FORTRAN compilers), there is still hope.

  If you are on a Unix system, you can use a program called `unbuffer` which is provided as part of a popular package called Expect.[5] Expect is actually a version of Tcl that's been souped up to handle interactive programs. It lets you write scripts to control these programs, so you can automate things that would normally require a human operator.

5. Don Libes, *Exploring Expect*, O'Reilly & Associates, 1995.

The `unbuffer` program executes another program, and then fools the program into thinking that it is being run interactively from a terminal. Remember, most programs won't buffer their output in this mode, so the buffering problem is solved.

For example, let's revisit the `bufdemo` program that we used to demonstrate the buffering problem a moment ago. When we run it like this, it buffers the output:

```
$ bufdemo | cat
```

Without changing the source code, we can run it like this, and the buffering stops:

```
$ unbuffer bufdemo | cat
```

Suppose you want to read the output from `bufdemo` within a Tcl program. You might open a pipe to the program and read the output like this:

```
set fid [open "|bufdemo" "r"]
set line [gets $fid]
```

But `bufdemo` buffers its output, so it would take 1.5 minutes for the first chunk of output to become available for reading. Again, you can solve this problem by adding the `unbuffer` program to the pipe, like this:

```
set fid [open "|unbuffer bufdemo" "r"]
set line [gets $fid]
```

Each line of output will be available as soon as it is written from `bufdemo`.

7.5 Bidirectional Pipes

So far, we've read from pipes in one example and written to pipes in another. These were examples of *unidirectional pipes*, since the output was either coming from or going to another process. Now let's look at *bidirectional pipes*, where we can both read from and write to a process on the same file descriptor.

For example, suppose we're writing a calculator-style application, and we need more precision than Tcl's built-in math library could provide. On many Unix systems, we can run the binary calculator program `bc`, which handles arbitrary-precision arithmetic. We can execute this program within our Tcl/Tk application, and talk to it through a bidirectional pipe. We can write arithmetic commands to its standard input and read the results back from its standard output.

7.5.1 Buffering Problems

In the previous section, we showed how to work around the problems that arise when a program buffers its output. With bidirectional pipes, there are two programs producing output, as shown in Figure 7.6. One is our Tcl/Tk application, which writes a string like "`2+2`" to the `bc` program. The other is the `bc` program, which writes back a result string like "`4`".

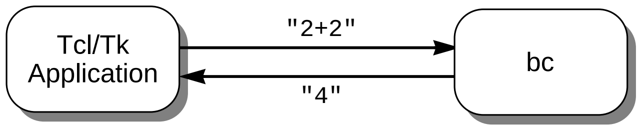


Figure 7.4. With bidirectional pipes, there are two sources of output, and therefore, two sources of buffering problems.

If either of these programs buffers its output, our calculator won't work. For example, suppose that the Tcl/Tk application writes out the string "`2+2`" and then tries to read the response from `bc`. If the Tcl/Tk application buffers its output, then `bc` will never receive the string, so it will sit idly, waiting for input. Meanwhile, our Tcl/Tk application will sit idly, waiting for the response.

We can solve half of the problem by making sure that our Tcl/Tk application does not buffer the output to `bc`. As we explained earlier, we can do this by setting the buffering mode for the pipe, like this:

```
set bc [open "|bc" "r+"]
fconfigure $bc -buffering line
```

So each line of output from the Tcl/Tk application will be flushed automatically.

Now suppose that the `bc` program buffers its output. It may receive a string like "`2+2`", and write out the result "`4`", but this result will sit in its buffer. Our Tcl/Tk application will sit idly, waiting for the response, and the `bc` program will sit idly, waiting for more input.

Some implementations of `bc` do indeed buffer their output. If you happen to have one, then your options are rather limited. As we discussed in Section 7.4.2, you can do one of the following:

- Obtain the source code for the `bc` program and modify it to flush after printing each result.
- Use Expect to work around the problem. Unfortunately, the simple `unbuffer` solution that we mentioned earlier won't work in this case. The `unbuffer` program blocks the standard input to the target program, so in this example, it would block incoming strings like "`2+2`".

  In a case like this, you should use Expect instead of the normal Tcl/Tk distribution. Exactly how you do this is outside the scope of this book, but it works something like this. Expect is just like Tcl/Tk, but it has extra commands to deal with interactive programs. You can use Expect's `spawn` command to execute the `bc` program from within your application. Then, you can use its `send` command to send a string like "`2+2`", and use its `expect` command to look for the result string "`4`".

We have included a simple Expect program in the file *fixes/bc* in the software that accompanies this book. This program makes the version of `bc` from the Free Software Foundation work for the examples shown in this chapter. It removes the header lines that this version of `bc` normally prints, and it makes sure that output is not buffered.

7.5.2 Writing and Reading

Now that our buffering problem is solved, let's try a simple example. We can start up Tcl and enter commands interactively to talk to the `bc` program, like this:

```
   % set bc [open "|bc" "r+"]
⇒  file5
   % fconfigure $bc -buffering line
   % puts $bc "2+2"
   % gets $bc
⇒  4
```

So far, so good, but now let's consider what will happen when we send an expression that returns a large result. Try starting `bc` by hand and enter an expression with a large exponent, like this:

```
$ bc
727^210
83603850580533190307486821101809589657323546469698265996423983062139\
98799546732779041712223034222327265648008832574513679609711177139023\
...
28066899857635655953102554062043350264667475060611670748010950572265\
888510191206304132167288269696997520892158042986708901649
```

As you can see, `bc` splits the result into multiple lines, and indicates that the line is being continued with a trailing backslash on each line. So reading the result back from `bc` may take more than a single `gets` command. We can handle this by writing a small procedure to read the output from `bc`, concatenating lines that are terminated with a backslash:

```
proc getbc {bc} {
     set answer ""
     set line "\\"
     while {[string last "\\" $line] >= 0} {
          append answer [string trimright $line "\\"]
          set line [gets $bc]
     }
     append answer [string trimright $line "\\"]
     return $answer
}
```

You can start up Tcl and open the `bc` program as we did a moment ago, to test the new procedure. This time, instead of using the `gets` command to read the result, use `getbc`. It returns the result as a single (very long!) number, which we have abbreviated below:

```
% puts $bc "727^210"
```

```
   % getbc $bc
⇒ 8360385058053319030748682110...58042986708901649
```

7.5.3 Graphical Interface

Now that our Tcl/Tk application can talk to the `bc` program, we can build a calculator like the one shown in Figure 7.5. We showed how to create and align the widgets in the calculator when we talked about the `grid` command in Section 2.2.2. Now we'll discuss how it actually works.

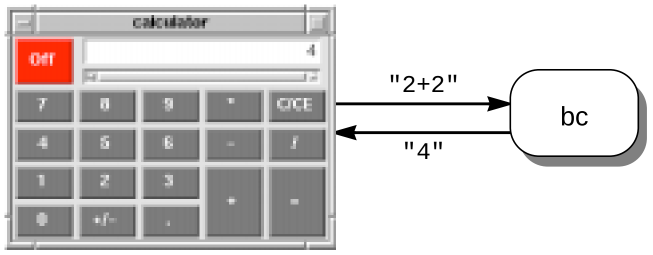


Figure 7.5. Graphical interface for the `bc` calculator program.

When you press a number key or the decimal point, we add that character to the string shown in the readout. When you press an operator key like `+`, `-`, `*` or `/`, we consider that operation to be pending, and we start to build another operand string. Finally, when you press the = key, we write the entire expression out to the `bc` program, read back the result, and display it in the readout window. If you'd like to see exactly how this is done, you can look at the complete application in the file *apps/calc*, in the source code that accompanies this book.

But the important point is this: You can take an existing program like `bc` and dress it up with a spiffy graphical interface. You can use the program as-is, and simply talk to it through a bidirectional pipe. This is a powerful technique for extending many programs. This includes the programs that you're unable to modify, and the legacy code that you don't want to modify.

7.6 Client/Server Architectures

In the last section, we stumbled onto an important idea: We used two cooperating programs to implement a single application. We started with the `bc` program, which can do arbitrary-precision arithmetic but has a simple, command-line interface. This program is called the *back-end process* or the *engine*, since it does all of the real work in the application. Then, we added a Tcl/Tk program providing a calculator-style interface. This program is called the *front-end process* or the *interface*, since it interacts with the user.

For our simple calculator, there is a master/slave relationship between the front-end process (Tcl/Tk) and the back-end process (`bc`). When you start up a calculator, the Tcl/Tk front end starts up its own `bc` back end. When you turn off a calculator, the Tcl/Tk front end exits, causing the `bc` back end to terminate.

You can also have a single back-end process that serves many different front ends. For example, you may have a back end that provides access to a database. Each user could start up his own front-end process, connect to the back end, and query information from the database. Usually, this kind of back-end process runs continuously, waiting for new front-end processes to connect and request information. This kind of back-end process is called a *server*, and each front-end process is called a *client*.

7.6.1 Advantages

Using separate front-end/back-end processes has some advantages:

- You may want to run the front end and the back end on different machines. For example, you may have a graphical interface running on a personal computer (PC), talking to a database server running on a larger centralized system. This is an example of traditional client/server programming.

  The emergence of the World Wide Web (WWW) has changed things a bit. These days, you can run your client program from a web browser[6] and talk to a remote server across the Internet.
- You may not want to (or be able to) modify the back-end program. For example, you may have a commercial program or some legacy code that you simply want to wrap with a graphical interface.

There are also advantages related to design and coding:

- This approach is good for a programming team. It lets programmers work independently on cooperating programs, instead of integrating their code into a single, gigantic program. This is especially useful during the testing and debugging phases. Normally, a piece of code in one subsystem can corrupt (“step on”) another subsystem’s data area, causing it to fail unexpectedly. But since the front-end and back-end code are in different processes, there are no unexpected interactions between the two, and no finger-pointing about whose code might be causing a bug. You’ve already seen how this pays off in “A Tale of Two GUI Programmers” at the start of this chapter.

6. Tcl/Tk applications can be embedded within a web page. See details at: *http://www.sunlabs.com/research/tcl/plugin/*

Also, you can partition a problem into smaller domains. For example, your database expert can work on the back-end process, and your GUI expert can work on the front end. Neither one has to know much about the other problem domain. Instead, they communicate by passing simple strings back and forth. Even inexperienced developers can handle this.

- You can write several different front-end programs without modifying the back-end program. So you can have a graphical front end for general use, a batch-style front end for automated system updates, and so on. Also, you can write different front ends for different users, customizing the front end for their particular needs.
- There is a well-defined interface to the back-end process, so you get a testing interface at no extra cost. This makes it easy to write test programs to exercise the back end. And if the front end encounters an unexpected error in the back end, the problem is easy to reproduce.
- The front-end and back-end programs can be written in different languages.

7.6.2 Disadvantages

Depending on your application, there are a few drawbacks to consider:

- There may be extra work involved in designing and maintaining the interface for the back-end process.
- There can be a performance penalty from crossing processing boundaries. This is usually not the case for a "traditional" graphical interface, since a human being's perception of time is so much slower than a computer's. Adding 10 milliseconds to a typical transaction won't be noticed by anyone but Superman or The Flash.

  However, if you need to download a lot of data from the back end, or if you need to send a lot of queries back and forth, you may be better off using a single process. For example, suppose you're using a client to plot data from the server. If the server generates a few hundred points, you may not notice the delay. But if it generates a few million points, you certainly will.
- There is an extra program to manage in configuration control, and an extra process to keep track of while executing. This by itself shouldn't preclude the front-end/back-end approach, but it helps to take this into account when planning your system.

If you're wondering whether to use a single program or the front-end/back-end approach, there is no simple answer. Like so many other things in software design, you have to weigh the trade-offs and make the right choice for the project at hand.

7.6.3 A Simple Server

Now let's see how to create a simple client/server application. Suppose we create a simple math server that can perform two functions: addition and subtraction. For convenience, we'll write both the client and the server in Tcl, but you can write either program in any other language as well. So you can use a client written in Tcl with a server written in C, or a client written in C++ with a server written in Tcl.

7.6.3.1 How It Works

Our simple math server will perform two functions:

- `add` *x y* ....returns *x+y*
- `subtract` *x y*....returns *x-y*

Each request will be sent in the form of a string consisting of a command (either `add` or `subtract`) followed by two parameters (the numbers to be added or subtracted). The command and the parameters will be separated by spaces.

Figure 7.6 shows a request and the response for a typical transaction with the math server. As a first pass, we'll have the client and the server communicate through a bidirectional pipe, just as we did in Section 7.5. Later we'll show how use a socket connection so that the two processes can live on different machines in a network. But even though the communication channel may change, the requests and responses sent back and forth will stay just the same.

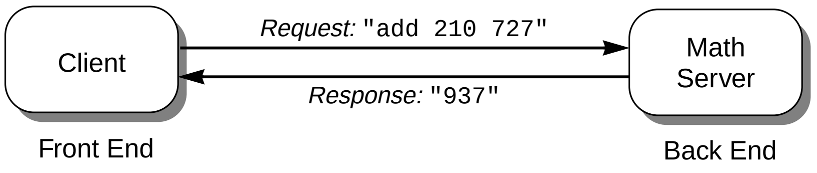


Figure 7.6. Math server, first pass.

7.6.3.2 A Brute Force Implementation

Our server will read a line from its standard input, handle the `add` or `subtract` operation, and print the result. It will continue doing this as long as input is available.

How should we implement this? Well, as Ken Thompson said, "When in doubt, use brute force."[7] With this in mind, let's take a first cut at our math server:

```
while {[gets stdin request] != -1} {
    set cmd [lindex $request 0]
    switch -- $cmd {
```

7. Ref Bentley, Programming Pearls, "Bumper Sticker Computer Science" chapter.

```
            add {
                if {[llength $request] == 3} {
                    set parm1 [lindex $request 1]
                    set parm2 [lindex $request 2]
                    set result [expr $parm1 + $parm2]
                    puts $result
                } else {
                    puts "error: add should have 2 parameters"
                }
            }
            subtract {
                if {[llength $request] == 3} {
                    set parm1 [lindex $request 1]
                    set parm2 [lindex $request 2]
                    set result [expr $parm1 - $parm2]
                    puts $result
                } else {
                    puts "error: subtract should have 2 parameters"
                }
            }
            default {
                puts "error: unknown command: $cmd"
            }
        }
    }
```

The `gets` command reads a line from standard input, saving it in the variable `request`. It returns the number of characters in the line, or `-1` at the end-of-file condition. So we use the `while` loop to continue reading until there is no more input.

Each request line has an operation and its arguments, all separated by spaces. As far as Tcl is concerned, this is like any other list, so we can use the `lindex` command to pick it apart. The first element (at index 0) is the operation, which we store in a variable called `cmd`.

Now, we simply look at the operation and handle it accordingly. We use a simple `switch` command to handle the various cases. For the `add` operation, we get the next two elements in the request, add them together, and print the return result. For the `subtract` operation, we get the next two elements, subtract them, and again, print the return result. If anything goes wrong, we print an error message as the result. Since we are printing to standard output, each line is automatically flushed. So we don't have to worry about output buffering in this case.

Whenever you use a `switch` command, you should always add the `--` argument to guard against unexpected errors. You see, the `switch` command has some optional flags that control its behavior. For example, if you say:

```
switch -regexp "foo" {
    f.* { ... }
}
```

then the `switch` command will interpret patterns like `f.*` as regular expressions. Now suppose that a math client accidentally sends a request like "`-3 + 12`". Our server will think that `-3` is the operation, and it will feed this to the switch. If we didn't include the `--` argument, the switch would look like this:

```
switch -3 {
    add { ... }
    subtract { ... }
    ...
}
```

Since the argument after the `switch` command starts with a dash, the switch would treat it as an option like `-regexp` and return the following error:

```
bad option "-3": should be -exact, -glob, -regexp, or --
```

By adding the `--` argument, we're telling the switch that there are no more control flags, and the next argument that follows is the string that we're trying to match.

7.6.4 A Simple Client

Now let's turn our attention to the front-end process. We'll build a simple client like the one shown in Figure 7.7. It has entries for two numbers, with + and - buttons between them. When you press either of these buttons, it sends a request to the server, and displays the result in a label on the right-hand side.

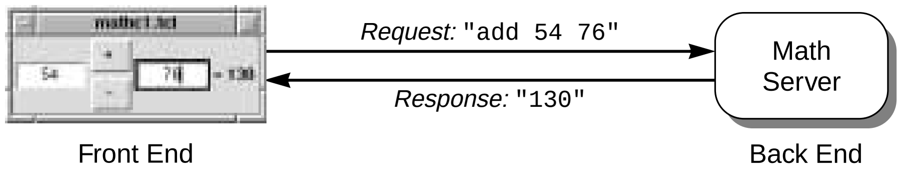


Figure 7.7. Talking to the math server.

The widget code in this client interface is straight-forward. It looks like this:

```
entry .x
pack .x -side left
frame .op
pack .op -side left
button .op.add -text "+" -command do_add
pack .op.add -fill both
button .op.sub -text "-" -command do_subtract
pack .op.sub -fill both
```

```
entry .y
pack .y -side left
label .result -text ""
pack .result -side left
```

As we said earlier, we'll start by using a bidirectional pipe to communicate with the server. So the code for this client looks just like the code that we saw in Section 7.5.2. We connect to the server and handle the + and - buttons like this:

```
set backend [open "|tclsh maths1.tcl" "r+"]
fconfigure $backend -buffering line

proc do_add {} {
    global backend
    set x [.x get]
    set y [.y get]
    puts $backend "add $x $y"
    gets $backend num
    .result configure -text "= $num"
}

proc do_subtract {} {
    global backend
    set x [.x get]
    set y [.y get]
    puts $backend "subtract $x $y"
    gets $backend num
    .result configure -text "= $num"
}
```

We open a bidirectional pipe to the math server, storing the file handle in a variable called `backend`. Remember, the math server was written in Tcl, and the code is stored in a file called *maths1.tcl*, which we assume to be in the current directory. So we use the command `tclsh maths1.tcl` to start the server. We set the output buffering for this pipe to `line` mode, so each request line that we print to the pipe will be flushed automatically. This avoids any buffering problems, as we discussed earlier in Section 7.4.2.

The + button invokes the `do_add` procedure, and the - button invokes `do_subtract`. These two procedures are very much alike. They both query the numbers from the entry widgets `.x` and `.y`, and then print a request to the server. They both read the result and display it in the label called `.result`. The only difference is that `do_add` prints an `add` request, and `do_subtract` prints a `subtract` request.

7.6.5 Smarter Parsing

Now let's look at the server program more carefully. It works, but it's not particularly elegant. The bulk of this code (about 80%!) handles argument parsing, error checking, and error reporting. Had we written our back end in C or C++, the percentage would be about the same. The bottom line is this: If you write your servers with the brute force approach, you'll spend 80% of your time implementing a new command language for each server.

Did we just say *command language*? Don't we have a nice *Tool Command Language*[8] sitting around? Of course we do. We can use Tcl itself to handle the request parsing for our server. This will dramatically reduce the amount of code in the server, and give us error checking for free!

For example, we could rewrite our math server like this:

```
proc add {x y} {
    return [expr $x+$y]
}
proc subtract {x y} {
    return [expr $x-$y]
}

while {[gets stdin request] != -1} {
    if {[catch $request result] != 0} {
        puts "error: $result"
    } else {
        puts $result
    }
}
```

This time, we defined `add` and `subtract` as procedures in the Tcl interpreter. The `add` procedure takes two arguments, and returns their sum. The `subtract` procedure takes two arguments, and returns their difference.

We read each line of input just as we did before. But this time, instead of parsing it ourselves, we simply execute each request as a Tcl command. The `catch` command executes the request string and returns a status code indicating success or failure. That way, we can trap errors and handle them gracefully.

If anything goes wrong, `catch` returns a non-zero status code, along with an error message in the `result` variable. In that case, we return a result string that says `error:` followed by the error message. If the command is successful, `catch` returns zero, along with a valid result in the `result` variable. In that case, we return the result string directly.

Of course, it was trivial for us to use the Tcl interpreter in our math server, since we were using Tcl already. But it's not much more trouble to do the same thing in C or C++. With just a few lines of code, you can drop a Tcl interpreter into your program and add Tcl commands to handle the important operations.[9] When all is said and done, you'll probably decrease your code size, thereby decreasing your maintenance costs.[10]

8. That's the acronym behind Tcl.

9. For details, see: John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

When you think parsing, think Tcl,
the Tool Command Language!

7.6.6 Safer Parsing

In the last section, we built a math server that was smaller, faster, and better. What more could we ask? Well for starters, we could ask for something *safer*.

Suppose an unfriendly programmer puts this bit of code in his front-end program:

```
button .b -text "Add" -command "send $backend {exec rm -rf *}"
```

When you press the *Add* button, it sends the command `exec rm -rf *` to the server, which happily executes the command. On Unix platforms, this will delete all of the files on your file system! Remember, the `catch` command will execute *any* command in the Tcl interpreter, including `exec`.

You may trust your programmers not to do something nefarious like this, but what if you have an unfriendly user? Suppose some user enters the following string in one of the number fields:

```
[exec rm -rf *]
```

Our client would package up a command that looks something like this:

```
add [exec rm -rf *] 17.5
```

and send it to the back end to be executed, again with disastrous results. This is a serious security risk, and it must be fixed. Fortunately, Tcl has a simple solution.

7.6.6.1 Safe Interpreters

Within each Tcl interpreter, you can create sub-interpreters called *slave* interpreters, and execute code within them. There is a special kind of slave interpreter called a *safe interpreter.* It has most of the usual Tcl commands, but none of the dangerous commands like `exec` and `open`. So you can use a safe interpreter to execute arbitrary code without worrying that it will harm your environment.

You can create a safe interpreter and have it execute some code like this:

```
set parser [interp create -safe]
set num [$parser eval {expr 2+2}]
```

The `interp create` command creates a safe interpreter and returns its name, which we store in the variable `parser`. We execute some code simply by using this name to access the interpreter. In this case, we're telling it to evaluate the `expr 2+2` command, and we store the result in a variable called `num` in the main interpreter.

Each interpreter has its own set of commands and global variables. You can add new commands to a slave interpreter just by defining a few procedures. For example, we could define `add` and `subtract` procedures in the safe interpreter like this:

10. Ref Bill Gates story?

```
$parser eval {
    proc add {x y} {
        return [expr $x+$y]
    }
    proc subtract {x y} {
        return [expr $x-$y]
    }
}
```

Now that it understands the `add` command, we can use the safe interpreter to add two numbers like this:

```
set num [$parser eval {add 2 2}]
```

At this point, we have a safe interpreter that accepts commands like `add` and `subtract`, but rejects anything unsafe like `exec` or `open`. So we can go back and rewrite our math server to use a safe interpreter to handle all incoming commands. Our new math server looks like this:

```
set parser [interp create -safe]

$parser eval {
    proc add {x y} {
        return [expr $x+$y]
    }
    proc subtract {x y} {
        return [expr $x-$y]
    }
}

while {[gets stdin request] != -1} {
    if {[catch {$parser eval $request} result] != 0} {
        puts "error: $result"
    } else {
        puts $result
    }
}
```

We read each request just as we did before, but this time instead of evaluating a request directly within the `catch` command, we use the safe interpreter to evaluate it. We still catch the result, so we can report any errors.

Should some evil client send a destructive command, they will simply get an error message such as

```
invalid command name "exec"
    while executing
"exec rm -rf *"
    invoked from within
"add [exec rm -rf *]..."
```

7.6.6.2 Aliases

Let's digress for a moment, so we can explore another important feature of safe interpreters. Suppose we wanted to write an `add` procedure that supports arbitrary-precision arithmetic. We saw how to handle this with the `bc` program in Section 7.5.2. We simply open a pipe to the `bc` program and add the numbers as follows:

```
$parser eval {
    proc add {x y} {
        set bc [open "|bc" "r+"]
        puts $bc "$x + $y"
        flush $bc
        set result [gets $bc]
        close $bc
        return $result
    }
}
```

This procedure starts up the `bc` program, writes out the add instruction, reads back the result, and shuts down the `bc` program. This may not be the most efficient solution, but we're trying to keep things simple for the current discussion.

If you test out this new `add` command, you'll get the following error:

```
  % $parser eval {add 2 2}
⇒ invalid command name "open"
```

You see, the `add` procedure executes in the context of the safe interpreter, and there's no way to execute the `open` command in that interpreter. Although the `add` command itself is safe, we need to do something unsafe as part of its implementation.

There is a simple way to solve this problem. If a procedure like `add` is defined in the main interpreter, it will have access to all of the Tcl commands in that interpreter, including `exec` and `open`. So it will execute properly in that context. Instead of adding the procedure to the safe interpreter, you can add it to the main interpreter and create an *alias* in the safe interpreter that points to the real command. For example:

```
proc cmd_add {x y} {
    set bc [open "|bc" "r+"]
    puts $bc "$x + $y"
    flush $bc
    set result [gets $bc]
    close $bc
    return $result
}
$parser alias add cmd_add
```

This defines a procedure called `cmd_add` in the main interpreter, and then adds an alias called `add` in the safe interpreter. When you execute the `add` command in the safe interpreter like this:

```
$parser eval {add 2 2}
```

it transfers control to the real `cmd_add` procedure in the main interpreter, which adds the two numbers.

You can use aliases like this to control access to things in the main interpreter. You can also use aliases to provide safe versions of otherwise unsafe commands. For example, suppose you want to let clients use the `open` command, but you want to restrict their access to files in a temporary directory. You could write a procedure called `safe_open` in the main interpreter. It could examine each file name, making sure that it points to the temporary directory, and it could use the normal `open` command to open the file. Then, you could add an alias called `open` in the safe interpreter, which points to `safe_open` in the main interpreter. Clients would think that they were using the regular `open` command, but any attempt to access a file outside of the temporary directory would be blocked.

7.6.6.3 Using Tcl Commands to Store Data

You should always use a safe interpreter when you're evaluating commands from an outside source. This includes the commands in a data file, if you're using Tcl commands to express data. For example, we created a drawing program in Section 4.7 that saved a drawing as a series of `draw` commands. At the time, we showed how you could load a drawing via the `source` command. Now, let's revisit this example and see how to load a drawing properly, using a safe interpreter.

First, we create a safe interpreter to handle the `draw` commands:

```
set canvParser [interp create -safe]
```

We save the name of this interpreter in a global variable called `canvParser`, so we can access it later.

Next, we write a procedure called `canvas_load` which loads a drawing onto a canvas. It takes the name of a canvas and a script of `draw` commands, and it uses the safe interpreter to execute the script. It is implemented as follows:

```
proc canvas_load {win script} {
    global canvParser
    $win delete all
    $canvParser alias draw canvas_parser_cmd $win
    $canvParser eval $script
}
```

We start by deleting all of the items on the canvas, to prepare for the new drawing. Then, we configure the safe interpreter to recognize the `draw` command. Each `draw` command creates an item on the canvas. But the canvas isn't available in the safe interpreter. It belongs to the main interpreter, so we create an alias for the `draw` command that transfers control to the `canvas_parser_cmd` procedure in the main interpreter. This procedure will have no problem accessing the canvas.

Notice that when we defined this alias, we added the `$win` argument after `canvas_parser_cmd`. This makes the argument part of the translation. Suppose that `$win` contains the canvas name `.drawing`. If you execute a `draw` command in the safe interpreter that looks like this:

```
draw rectangle 79.0 24.0 256.0 196.0 -fill red
```

then the alias will trigger a call to `canvas_parser_cmd`, like this:

```
canvas_parser_cmd .drawing rectangle 79.0 24.0 256.0 196.0 -fill red
```

As you can see, the canvas name is automatically included as the first argument on the command line. The rest of the arguments follow exactly as they appear in the `draw` command.

The `canvas_parser_cmd` procedure is implemented like this:

```
proc canvas_parser_cmd {win args} {
    eval $win create $args
}
```

This looks a lot like the `draw` procedure that we defined in Section 4.7.9. It passes the item description contained in `$args` to the canvas `create` operation, creating the item on the canvas.

Now that we have the `canvas_load` procedure, we can use it within our drawing program to load a drawing. First, we add an *Open...* command to the *File* menu, like this:

```
.mbar.file.m add command -label "Open..." -command draw_open
```

When you select the *Open...* command, it calls the following procedure to load the drawing:

```
proc draw_open {} {
    global env
    set file [tk_getOpenFile]
    if {$file != ""} {
        set cmd {
            set fid [open $file r]
            set script [read $fid]
            close $fid
            canvas_load .drawing $script
        }
        if {[catch $cmd err] != 0} {
            notice_show "Cannot open drawing:\n$err" error
        }
        draw_fix_menus
    }
}
```

This procedure uses `tk_getOpenFile` to get the name of the input file. This pops up a file selection dialog, letting you navigate the file system and select a file. If you press the *Cancel* button, then `tk_getOpenFile` returns a null string, and the `draw_open` procedure does nothing. Otherwise, `tk_getOpenFile` returns the name that you selected for the drawing file.

To load the drawing, we need to open the file, read the script, close the file, and call `canvas_load` to evaluate the script. Any of these operations could fail, so we wrap them up as a small script and execute them using the `catch` command. If we catch any errors, we use the `notice_show` procedure developed in Section 6.4 to display the error message in a notice dialog. We also use a procedure called `draw_fix_menus`, which we won't show here. It just checks to see if any items in the drawing are selected, and then enables or disables some commands in the *Edit* menu.

Tcl commands are a powerful way of expressing data. As long as you use a safe interpreter to load the data, you can avoid any mishaps. In the current example, if you try to load a drawing file that looks like this:

```
draw rectangle 79.0 24.0 256.0 196.0 -fill red
exec rm -rf *
```

the `canvas_load` procedure will simply complain that `exec` is an invalid command, and the `draw_open` procedure will notify the user that the operation failed. Your files will remain intact, in spite of someone's best effort to clobber them!

7.6.7 Asynchronous Communication

So far, all of the interactions between our front-end and back-end programs have been *synchronous*. The front end sends a request string, and it waits for the response string, as shown in Figure 7.8. While it is waiting, execution in the front-end program is blocked. If the front end has a graphical user interface, it won't be responsive to the mouse or the keyboard, and its windows won't refresh themselves properly when you cover and uncover them. If the back end doesn't respond quickly, most users will think that the front end has mysteriously locked up.

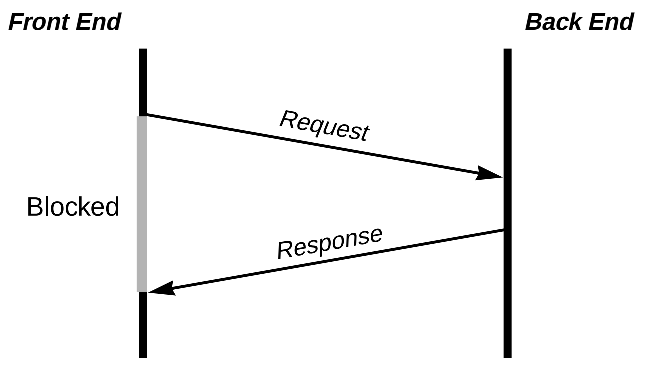


Figure 7.8. For a *synchronous* call, the front end sends a request and does nothing else until the back end sends a response.

You can avoid this by using an asynchronous communication scheme, as shown in Figure 7.9. In this scheme, the front-end program sends a request, and then continues execution. While it is waiting for the response, it might spin the cursor, or it might display an *Abort* button that you can use to cancel the request. Eventually, the back-end program sends a response, and the front end reads it and handles it at its first opportunity.

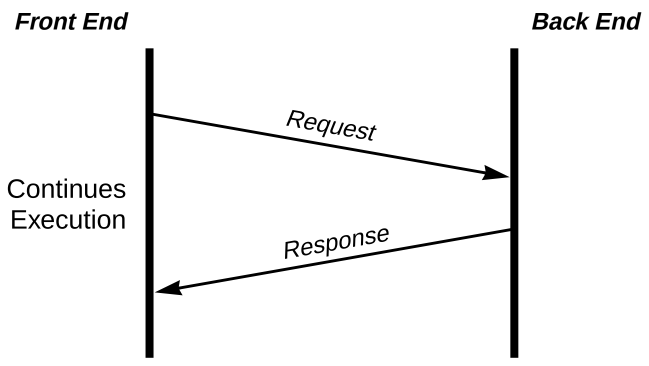


Figure 7.9. For an *asynchronous* call, the front end sends a request and continues execution. When the back end eventually sends the response, the front end handles it.

For this scheme to work, we can't have the front end send a request and read the response right away, like this:

```
puts $backend "add $x $y"
gets $backend num
```

Instead, we need to make it listen for messages that might appear at any time. We ran into the same problem when we were listening for output from the `pppstats` program in Section 7.2, and we can use the same solution here. We simply use the `fileevent` command to determine when the connection to the back end is readable. For example:

```
set backend [open "|tclsh maths4.tcl" "r+"]
fconfigure $backend -buffering line
fileevent $backend readable \
    "front_handler $backend $parser"

proc front_handler {fd parser} {
    if {[gets $fd request] < 0} {
        catch {close $fd}
        notice_show "Lost backend" error
    } elseif {[catch {$parser eval $request} result]} {
        notice_show $result error
    }
}
```

When a message arrives from the back end, the `front_handler` procedure is called to handle it. This procedure reads a line of input. If for some reason the back-end process has terminated, the `gets` command will return `-1`, indicating an end-of-file condition. In that case, we close the connection (ignoring any errors that we might encounter) and we let the user know that the back end has died. We use the `notice_show` procedure developed in Section 6.4 to display the message "Lost backend".

Otherwise, the `gets` command will return the number of characters in the request line. In that case, we try to evaluate the incoming message, and if something goes wrong, we use `notice_show` to notify the user.

We use a safe interpreter, which we refer to as `$parser`, to interpret all incoming messages in the front end. We'll see how to write the code for this in a moment, but first, we'll explain why this is needed.

You see, messages can arrive at any time from the back end, so we don't know *a priori* what to expect. When we send a command like `add 2 2`, it isn't enough to get back a response like `4`. By the time we receive that response, it may not be obvious what `4` means. So instead of having the back-end send back a data value like `4`, we'll modify it to send back a Tcl command like `show_result 4`. This tells us what to do with the result.

This scheme also lets the back-end program send notification messages to the front end. For example, if two clients are connected to a server, one client might want to be notified when another client changes some data. Since all incoming messages are in the form of commands, the server can "tell" the client that some data has changed. And since the communication is asynchronous, a message like this can be received and processed at any point—even when the front end is waiting for a response, as shown in Figure 7.10.

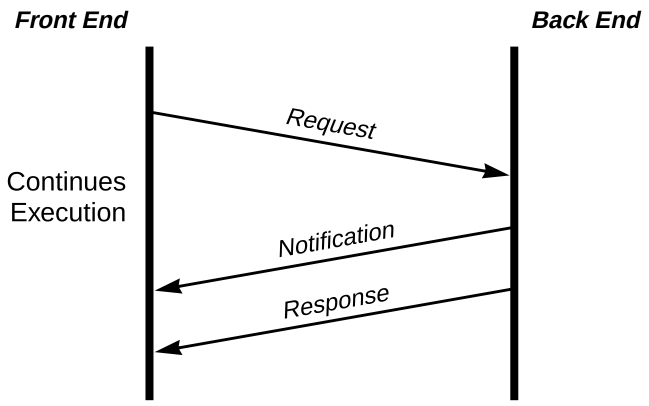


Figure 7.10. With asynchronous communication, the front end can receive a notification while it is waiting for a response.

Our math client is now set up to read incoming messages and interpret them, but one task remains: We need to determine the *protocol*, or the messages that are exchanged between the client and the server. There are several ways of doing this. We'll look at two of them in the following sections.

7.6.7.1 Pre-defined Callbacks

Let's assume that our math server will always respond to an `add` or `subtract` command by sending back the following string:

`show_result` *value*

where *value* is the result from the `add` or `subtract` operation. Different math clients could define the `show_result` procedure to do different things. One client might print out the value; another could save it in a log file. In the current example, we'll display the result in a label on the right-hand side of our graphical interface, as we saw earlier in Figure 7.7.

If the server encounters an error, it will send back the following message:

`error_result` *message*

Again, different clients could handle this in different ways, but in the current example, we'll simply display the *message* string in a notice dialog.

We can rewrite our math server to implement this new protocol as follows:

```
set parser [interp create -safe]

$parser eval {
    proc add {x y} {
        return [list show_result [expr $x+$y]]
    }
    proc subtract {x y} {
        return [list show_result [expr $x-$y]]
    }
}

proc back_handler {parser} {
    if {[gets stdin request] < 0} {
        exit
    } elseif {[catch {$parser eval $request} result] == 0} {
        puts $result
    } else {
        puts [list error_result $result]
    }
}

fileevent stdin readable "back_handler $parser"
vwait enter-mainloop
```

There are just a few places where the code has changed, so we've highlighted them in bold type.

Our last version of the server handled requests in a synchronous fashion. In this version, we changed the server to work asynchronously using the pattern described in the last section. We set up a file event on standard input, so that when input arrives, the procedure `back_handler` will be called to handle it. This procedure reads each request, executes it in a safe interpreter, and prints out the result.

In order to receive file events, the program must be executing in the event loop. If you use `wish` to execute the server program, this happens automatically at the end of your script. However, it is better to use `tclsh` to execute the server program, so that it can run in the background, without a window on the desktop. But `tclsh` doesn't drop into the event loop automatically. You have to force execution into the event loop using the `vwait` command. The `vwait` command waits for a variable to change. In this case, the variable `enter-mainloop` will never change, so the server will sit in the event loop as long as it continues to run.

In the last version of the server, we returned an error message when something went wrong. Now, we return `error_result` followed by the error message. You might be tempted to use double quotes to build the response string, like this:

```
puts "error_result $result"
```

But if we did this, it would probably cause an error in the client. Remember, the client will expect the `error_result` command to have a single argument. If the message in `$result` is something like `invalid command name "foo"`, then using double quotes will produce a response like:

```
error_result invalid command name "foo"
```

This looks like we're passing four arguments to `error_result`, so the client will complain.

Instead, we use the `list` command to build the return string. The `list` command automatically adds the braces needed to keep the error message together as a single argument. It produces a response string that looks like this:

```
error_result {invalid command name "foo"}
```

The client can interpret this response string correctly.

Similarly, in the previous version of the `add` and `subtract` procedures, we returned a number. Now we return `show_result` followed by a number. Again, we could have used double quotes to build the return string, like this:

```
return "show_result [expr $x+$y]"
```

But instead, we use the `list` command to build the return string. The `list` command isn't really necessary in this case, since the `expr` command won't return a result with spaces in it. But using the `list` command won't hurt, and it's a good habit to get into.

At this point, we've fixed the math server to respond to `add` and `subtract` commands with the new protocol. Now, we'll fix the math client to interpret the new protocol.

Since all of the incoming messages are Tcl commands, we can create a safe interpreter to handle them, as we discussed in Section 7.6.6. The safe interpreter will contain the two protocol commands. The `show_result` command will update a label on the graphical interface to show the result value, and the `error_result` command will pop up a notice dialog with an error message. Both of these procedures need to access widgets that exist in the main interpreter. So we need to define these procedures in the main interpreter and add aliases for them in the safe interpreter. For example:

```
proc cmd_show_result {num} {
    .result configure -text "= $num"
}
proc cmd_error_result {msg} {
    notice_show $msg error
}

set parser [interp create -safe]
$parser alias show_result cmd_show_result
$parser alias error_result cmd_error_result
```

When we execute `show_result` in the safe interpreter, it will call `cmd_show_result` in the main interpreter, which will change the text of the `.result` label to display the result. Likewise, when we execute `error_result` in the safe interpreter, it will call `cmd_error_result` in the main interpreter, which will call `notice_show` to pop up a notice dialog.

The client listens for incoming messages, and then uses this safe interpreter to evaluate them. We've seen the code that handles this. We described it in detail in Section 7.6.7, but we'll include again it here, so you can see how things fit together.

```
set backend [open "|tclsh maths4.tcl" "r+"]
fconfigure $backend -buffering line
fileevent $backend readable \
    "front_handler $backend $parser"

proc front_handler {fd parser} {
    if {[gets $fd request] < 0} {
        catch {close $fd}
        notice_show "Lost backend" error
    } elseif {[catch {$parser eval $request} result]} {
        notice_show $result error
    }
}
```

We simply set up a bidirectional pipe with the server program, and set up a file event to look for incoming messages. Whenever the pipe is readable, the `front_handler` procedure is called to handle the message. It reads a line of input and uses the safe interpreter to evaluate it. We use the `catch` command to look for errors, and if anything goes wrong, we report it in a notice dialog.

Finally, we need to modify the `do_add` and `do_subtract` procedures which send requests to the back end. We'll still use the `puts` command to send requests to the back end, but we don't need the `gets` command to read the response. We already have code in place to handle the response asynchronously. So these procedures now look like this:

```
proc do_add {} {
    global backend
    set x [.x get]
    set y [.y get]
    puts $backend "add $x $y"
}
proc do_subtract {} {
    global backend
    set x [.x get]
    set y [.y get]
    puts $backend "subtract $x $y"
}
```

Everything is in place to handle the new communication protocol. When you enter some numbers and press the + button on the client, it sends a message like this to the server:

```
add 2 2
```

The server evaluates this in its safe interpreter, and it sends back the message:

```
show_result 4
```

The client evaluates this in its safe interpreter, displaying the string "`= 4`" in the result label on the graphical interface. When you press the `-` button, it generates another request and another response, and so on.

7.6.7.2 Command Formatting with Templates

Having a specific protocol for requests and responses works just fine. But sometimes you need more flexibility in the responses returned by the back end. This becomes an issue when the back end returns several different values in a single response. For example, suppose we add a `divide` command to our math server that returns both the dividend and the remainder. Our standard `show_result` response won't handle this, since it was designed to return only a single value. You might be tempted to define a new response for the `divide` operation, like this:

`show_divide` *dividend remainder*

but adding new responses like this quickly gets out of hand.

Instead, we'll use a different approach: We'll have the client send a response template to the server, and have the server fill in the return values and send it back. For example, if the client sent a response template like:

```
show_result "= %d (remainder %r)"
```

then the server could replace `%d` with the dividend, replace `%r` with the remainder, and then send the string back. As far as the client is concerned, it sees an ordinary `show_result` message, which it already knows how to handle.

Let's see how this works for the simple `add` and `subtract` commands in our math server. Suppose we modify these commands to take the response template as a third argument, like this:

```
add x y response
subtract x y response
```

The server will add or subtract the numbers *x* and *y*, and then substitute the result in place of `%v` in the *response* template. So if we send the server a command like:

```
add 2 2 {show_result %v}
```

it will return the response:

```
show_result 4
```

Now suppose we want to change our client to display its results in a notice dialog. We could modify the client to send a different response template, like this:

```
add 2 2 {notice_show "The result is: %v"}
```

and the server will return a different response:

```
notice_show "The result is: 4"
```

Of course, the client may read this response, but it won't recognize the `notice_show` command unless we add `notice_show` to its vocabulary of incoming commands. We do this by adding an alias to the client's safe interpreter, like this:

```
$parser alias notice_show notice_show
```

When the `notice_show` message comes in, the client's safe interpreter will transfer control to the real `notice_show` procedure in the main interpreter, which will pop up a notice dialog with the result string.

Notice that we were able make this change to the client without changing anything in the server. We simply changed the client's response template, and we added to its vocabulary of incoming commands.

Using response templates has a few important advantages:

- It eliminates a lot of unnecessary protocol commands like `show_divide`, which would otherwise be added to handle the extra arguments. This simplifies the protocol between the client and the server, making the two programs easier to maintain.
- It improves the separation between the client and the server. The client sends the response templates over, and it interprets them when they come back. All of the response-related code resides within the client, so it is easier to maintain.
- It lets the client use the same server command in many different places, with many different results.

7.6.7.3 Percent Substitution

In the last section, we saw how response templates work. Now let's see how to modify the server to handle the substitutions.

As you'll recall, the server has a safe interpreter to handle incoming commands like `add` and `subtract`. We can modify these procedures to use a response template as follows:

```
set parser [interp create -safe]
$parser eval {
    proc add {x y cmd} {
        set num [expr $x+$y]
        set response [percent_subst %v $cmd $num]
        return $response
    }
    proc subtract {x y cmd} {
        set num [expr $x-$y]
        set response [percent_subst %v $cmd $num]
        return $response
    }
}
```

Both of these procedures get the response template via the `cmd` argument, and they substitute their result using a procedure called `percent_subst`. This procedure is a lot like the usual `regsub` command in Tcl, but it does something that regsub won't—it ignores characters like `&`, `\0`, `\1`, `\2`, ... that might be lurking in the result string. You can see why this is important in the following example.

Suppose you're building a server to access a marketing database. This database contains the names and phone numbers of thousands of people, and it identifies their long-distance telephone carrier. Now suppose that this server has a command to query the long-distance carrier:

`get_carrier` *customer response*

You pass in the *customer* name, the server accesses the database, finds the long-distance carrier, and substitutes the result into the `%c` field of the *response* string.

Now, suppose you send a request like this:

```
get_carrier "Hacker, Joe" {notice_show "long-distance: %c"}
```

and suppose the server uses `regsub` to substitute the result into the `%c` field, like this:

```
regsub -all %c $cmd $company response
```

The `-all` flag tells `regsub` to find all occurrences of the pattern `%c` in the template `$cmd` and replace them with the long-distance carrier name `$company`. The resulting string is stored in the `response` variable.

This works fine for company names like `MCI` and `Sprint`, but it fails miserably for a name like `AT&T`. The `regsub` command automatically replaces the `&` character with the pattern `%c`, so you get back the company name `AT%cT`! Using `regsub` directly to perform substitutions is an accident waiting to happen.

Instead, we use the `percent_subst` procedure to handle percent substitutions. It uses `regsub`, but it suppresses the normal substitutions for things like `&`, `\0`, `\1`, `\2`, and so on. It is implemented as follows:

```
$parser eval {
    proc percent_subst {percent string subst} {
        if {![string match %* $percent]} {
            error "bad pattern \"$percent\": should be %something"
        }
        regsub -all {\\|&} $subst {\\\0} subst
        regsub -all $percent $string $subst string
        return $string
    }
}
```

The first `regsub` command suppresses any dangerous characters in the substitution string `$subst`. It adds an extra `\` in front of each `\` or `&`, so the next `regsub` command will ignore them. A complete description of regular expressions is beyond the scope of this book.[11] But we'll explain briefly how this works. The first `regsub` command tries to match the pattern "`\\|&`" which means "either `\` or `&`". If it finds this pattern, it replaces it with "`\\\0`" which means "backslash (`\\`) followed by the matching character (`\0`)". It stores the resulting string back into the `subst` variable.

Having done this, we can use the second `regsub` command to substitute the `$subst` string in place of each percent field, without worrying about any of the dangerous characters. If `regsub` succeeds, it saves the updated string in the `string` variable. If it fails, it leaves the `string` variable alone. Either way, we simply return `$string` as the result of the procedure.

We use the `string match` command to make sure that the percent string really starts with a `%` sign. If not, we return a usage error.

Finally, we pass this whole procedure definition to the safe interpreter, so that the `add` and `subtract` commands will have access to it.

7.6.8 Handling Multi-line Requests

Not all of the commands that we pass to the front end or the back end will fit on a single line. For example, suppose we send our math server a command like:

```
add 2 2 {
    show_result %v
    notice_show "The server has responded."
}
```

11. See Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger, *The AWK Programming Language*, Addison-Wesley, 1988; and Don Libes, *Exploring Expect*, O'Reilly and Associates, 1995.

This should be treated as a single Tcl command, but the way the server is currently written, it will read only the first line, try to execute it, and return an error.

The server needs to keep reading until it has a complete command, and then execute that command. Fortunately, Tcl has the `info complete` command that we can use to determine if a string is syntactically complete. We can modify the `back_handler` procedure to use it as follows:

```
proc back_handler {parser} {
    global buffer
    if {[gets stdin request] < 0} {
        exit
    } else {
        append buffer $request "\n"
        if {[info complete $buffer]} {
            set request $buffer
            set buffer ""
            if {[catch {$parser eval $request} result] == 0} {
                puts $result
            } else {
                puts [list error_result $result]
            }
        }
    }
}
```

We use the global `buffer` variable to store each line of the incoming command. After appending each line, we check to see if the command is complete. If it is, we handle it just as we did before. If it is not, we return from `back_handler` and wait for another line.

Notice that before we actually evaluate the command, we transfer it from the buffer to the `request` variable, and we clear out the buffer. While the server is evaluating this request, it may handle more input, so we must prepare for this by resetting the buffer.

We can make a similar change to the `front_handler` procedure in our math client, so the client can handle multi-line responses as well.

7.6.8.1 Prompting for Commands

This same code comes in handy in another context: You can add an interactive shell to a Tcl/Tk program that is not being run interactively. Normally, when you execute `tclsh` or `wish` with a command script like this:

```
$ wish script.tcl
```

it won’t give you the usual command prompt. However, you can implement your own command prompt by adding the following code to the bottom of your script:

```
proc prompt {} {
    puts -nonewline "% "
    flush stdout
}
```

```
proc process {} {
    global buffer
    if {[gets stdin line] < 0} {
        exit
    } else {
        append buffer $line "\n"
        if {[info complete $buffer]} {
            set cmd $buffer
            set buffer ""
            catch {uplevel #0 $cmd} result
            puts $result
            prompt
        }
    }
}
fileevent stdin readable {process}
prompt
vwait enter-mainloop
```

This registers a file event for the standard input channel so that each line of input triggers a call to the `process` procedure. Of course, this will only work if our program enters the event loop to look for file events. The `wish` program will enter the event loop automatically, but `tclsh` won't. So to make this example work in both cases, we've included the `vwait` command to force execution into the event loop. The variable `enter-mainloop` should never change, so the program will stay in the event loop.

When we receive a file event, we read each line, append it onto the buffer, and check for a complete command. When we have one, we transfer it into the `cmd` variable, and then execute it.

Instead of using a safe interpreter in this case, we execute the command directly in the main interpreter. However, we use `uplevel #0` to execute the command at the global scope, outside of the `process` procedure. That way, if the command is something like `set x 0`, it will set a global variable called `x`, and not a local variable in the `process` procedure. We use `catch` to suppress any errors that we might encounter, then we print the result and prompt for another command.

7.7 Network Programming with Sockets

We've already seen how client/server programs can send commands back and forth to communicate. But so far, our client/server programs have been connected by a bidirectional pipe. In this section, we'll change the communication channel, replacing the bidirectional pipe with a *socket connection*. As far as your Tcl code is concerned, using sockets is a lot like using bidirectional pipes. But sockets have a couple of advantages:

- You can run the client (front end) and the server (back end) on different machines in the network. If your server is running on a system attached to the Internet, you can provide your service to computers all over the world.[12]
- One server can service several clients, as shown in Figure 7.11. Clients can share data and send messages to one another via the server.

Tcl provides its networking capabilities at a very high level, so you don't need to concern yourself with a lot of detailed programming to set up a socket connection. Once you have set up the connection, your client and server programs can communicate just as we described in the last section.

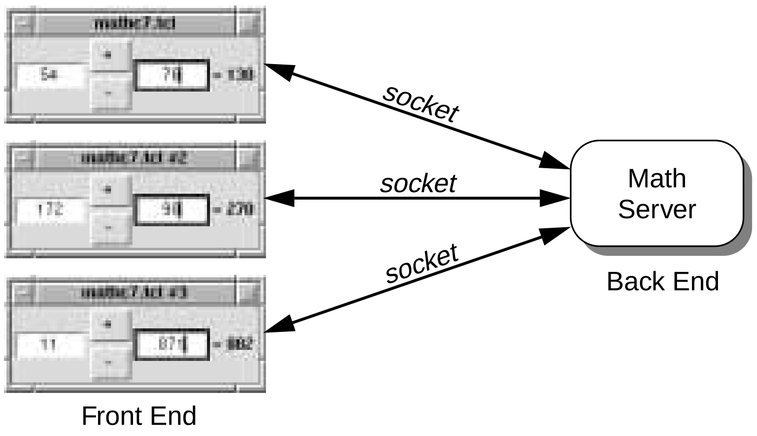


Figure 7.11. A networked server with several client processes. Each client has its own connection to the server.

7.7.1 Overview

Networked client/server applications work something like this:

1. A server process is started on a particular machine (called a *host*) on the network.

   The server opens a socket with a specific port number like `8111`, and then listens for clients to connect. You can choose any port number for the server, as long as it isn't being used by another application. Numbers in the range `8100-9999` are usually good choices.

12. Including the bad guys, so it's extremely important to use safe interpreters!

Normally, the server program runs until the host goes down. That way, clients can connect at any time. When the host comes back up, you can arrange for the server program to be restarted automatically. On Unix systems, you do this by adding the server program to a file called *rc.local,* which is usually in the */etc* directory.

2. A client program connects to the server.

The client connects to a particular host, using a host name like `frame.tcltk.com` or an Internet Protocol (IP) address like `128.92.104.10`. The client then asks for the server at a particular port, which is something like `8111`, as we decided in Step 1.

3. The server recognizes the client connection.

A special procedure is invoked within the server to handle each new client. This procedure usually sets up a file event to read input from the client. It may also allocate some resources for each client. For example, the server may open a separate data file for each client, to service its requests.

4. The client and server communicate by sending commands back and forth.

This works just as we described in Section 7.6.

5. Eventually, either the client or the server disconnects.

When this happens, the other side should detect the broken connection and close the socket. Server processes usually live longer than clients, so it is important that the server cleans up properly after each client. For example, if the server has opened a file for a client, it must close the file when the client disconnects. Otherwise, it will run out of file descriptors for future clients.

7.7.2 A Networked Server

You can create a server using the `socket` command like this:

```
socket -server command port
```

The `-server` option tells this command to listen for clients on a particular *port* number on the current host. When each client tries to connect, the server executes the *command* to recognize the new client.

Three parameters are automatically appended to *command*: the file handle for the client's connection, the IP address of the client's host, and the client's port number. Normally, the *command* parameter is simply the name of a procedure that takes three arguments. It usually looks something like this:

```
proc server_accept {cid addr port} {
    fileevent $cid readable "server_handle $cid"
    fconfigure $cid -buffering line
}
```

In this case, we have used the `fileevent` command to handle requests coming from the client. Whenever the client's connection becomes readable, the `server_handle` procedure will be called to read the request and handle it. Also, we have changed the output buffering for the client to `line` mode. That way, each response line that we send to the client will be flushed automatically. This avoids any buffering problems, as we discussed in Section 7.4.

Let's see how all of this comes together in a real example. We can rewrite our math server to use sockets as follows:

```
set parser [interp create -safe]
$parser eval {
    ...
}

proc server_accept {cid addr port} {
    fileevent $cid readable "server_handle $cid"
    fconfigure $cid -buffering line
}
proc server_handle {cid} {
    global parser buffer
    if {[gets $cid request] < 0} {
        close $cid
    } else {
        append buffer $request "\n"
        if {[info complete $buffer]} {
            set request $buffer
            set buffer ""
            if {[catch {$parser eval $request} result] == 0} {
                puts $cid $result
            } else {
                puts $cid [list error_result $result]
            }
        }
    }
}
socket -server server_accept 9001
vwait enter-mainloop
```

We start by creating a safe interpreter to handle incoming requests. We left out the code in the `$parser eval` command. This simply defines the `add` and `subtract` commands, as we saw earlier in Section 7.6.7.3.

Next, we define the `server_accept` procedure, which is called when each client connects. We also define the `server_handle` procedure, which is called when a request line arrives. This procedure looks a lot like the `back_handler` procedure shown in Section 7.6.8. It reads request lines from the socket and assembles them to form a complete command. When it has a complete command, it executes the command in a safe interpreter, and then writes the result back to the socket.

Next, we use the `socket` command to establish a server at port number `9001` on the current host.

Finally, we use the `vwait` command to enter the event loop, as discussed in Section 7.6.7.1. This causes the server to listen for clients to connect.

7.7.2.1 Testing the Server Manually

One advantage of passing text strings between client/server programs is that it makes the programs easy to test. If you want to verify that a server is running properly, you can simply connect to the server, send it some commands, and get back human-readable results. On Unix systems, you can use the `telnet` program to interact with a server. You simply give it the host name and the port number of the server that you want to talk to. For example, you could start up our math server and interact with it like this:

```
   $ tclsh maths7.tcl &
   $ telnet localhost 9001
⇒  Trying 127.0.0.1 ...
   Connected to localhost.
   Escape character is '^]'.
   add 2 3 %v
⇒  5
   add 2 3 "set x %v"
⇒  set x 5
   exec rm -rf *
∅  error_result {invalid command name "exec"}
```

7.7.3 A Networked Client

Let's rewrite our math client to use sockets, so it can talk to the new math server. Again, most of the client code will remain the same—we need only change the communication channel that we use to talk to the server.

```
proc client_handle {sid} {
    global backend parser buffer
    if {[gets $sid request] < 0} {
        catch {close $sid}
        set backend ""
        notice_show "Lost connection to server" error
    } else {
        append buffer $request "\n"
        if {[info complete $buffer]} {
            set request $buffer
            set buffer ""
            if {[catch {$parser eval $request} result] != 0} {
                notice_show $result error
            }
        }
    }
}
```

```
proc client_send {args} {
    global backend
    if {$backend != ""} {
        puts $backend $args
    }
}

set sid [socket localhost 9001]
fileevent $sid readable "client_handle $sid"
fconfigure $sid -buffering line
set backend $sid
```

The `client_handle` procedure is a lot like the `back_handler` procedure shown in Section 7.6.7.3. It reads response lines from the server, and assembles them to form a complete command. When it has a complete command, it executes the command in its safe interpreter. Remember, the server will send back commands like `show_result` or `error_result`. Executing these commands will cause the client to display its results.

We use the `socket` command to connect to the server program. For this simple example, we assume that the client and the server are running on the same machine, so we use `localhost` as the server's host name. If the server were running on a different host, we would replace `localhost` with the name of that machine.

If for some reason we can't connect to the server, the `socket` command will fail, and the client program will terminate. We could use the `catch` command to handle the error more gracefully, but in this simple example, we didn't bother.

If the connection is established, the `socket` command returns a file handle that we can use to communicate with the server. We should configure this connection just as we did on the server side. We set up a file event to handle response lines coming from the server, and we set the output buffering to `line` mode, so that each request line that we send to the server will be flushed automatically. Finally, we save the file handle in a global variable called `backend`, so we can use it throughout the program.

We've defined a procedure called `client_send` that makes it easy to send requests to the server. You simply append the request as arguments to this command. For example, we might use this to handle the `add` request as follows:

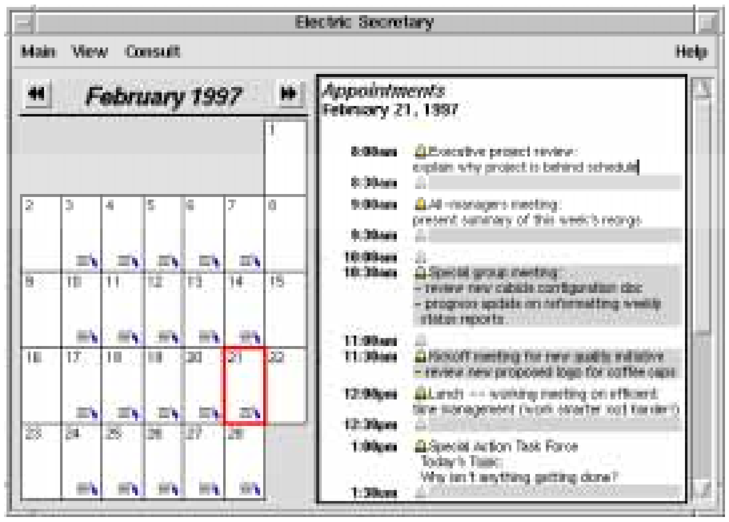
```
proc do_add {} {
    set x [.x get]
    set y [.y get]
    client_send add $x $y {show_result %v}
}
```

When you press the + button on the client interface, it invokes this procedure, which extracts the numbers from the two entry fields and then sends an `add` request off to the server. The server sends back a `show_result` command, which causes the client to display the result.

7.8 The Electric Secretary—A Case Study

Let's look at a case study that illustrates many of the concepts that we've studied so far. The application, which is shown in Figure 7.12, is a multi-user scheduling/calendar program called the *Electric Secretary.* Its features include:

- Standard perpetual calendar
- Day-by-day appointment minder
- Configurable alarms for appointments
- Group calendar management
- Options for colors, alarm characteristics, and work groups



```
 8:00am  Executive project review:
         explain why project is behind schedule
 8:30am
 9:00am  All-managers meeting:
         present summary of this week's reorgs
 9:30am
10:00am
10:30am  Special group meeting:
         - review new cubicle configuration doc
         - progress update on reformatting weekly
           status reports
11:00am
11:30am  Kickoff meeting for new quality initiative
         - review new proposed logo for coffee cups
12:00pm  Lunch -- working meeting on efficient
         time management (work smarter not harder!)
12:30pm
 1:00pm  Special Action Task Force
          Today's Topic:
          Why isn't anything getting done?
 1:30pm
```

Figure 7.12. Dilbert's boss uses the *Electric Secretary.*

This application uses our client/server architecture, as shown in Figure 7.13. A networked server keeps the appointment data in a centralized location. Clients connect to the server and display the appointment data in a graphical user interface. Each client has certain configuration options called *preferences* that are stored in a file in the user's home directory. So each user can configure the client program to customize things like colors and alarm characteristics.

The server program is about 200 lines long. The client program is about 850 lines long, not counting the components like the calendar and the appointment editor that we

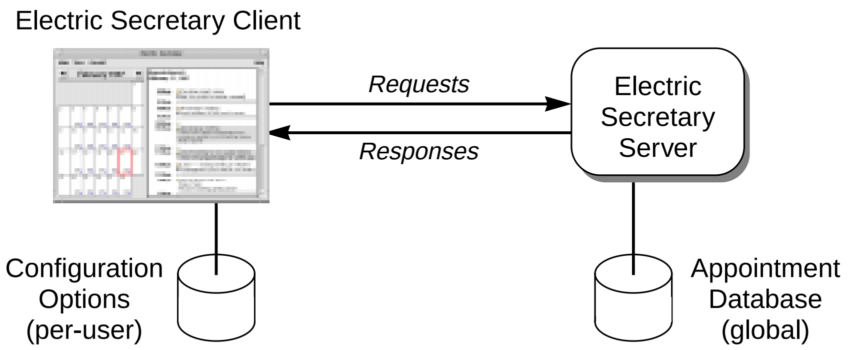

Electric Secretary Client
Requests
Responses
Electric Secretary Server
Configuration Options (per-user)
Appointment Database (global)

Figure 7.13. Electric Secretary client/server architecture. Multiple clients can be attached to the server at the same time.

developed in previous chapters. We won't go into the details of this code. Instead, we'll focus on the overall architecture, and describe the communication between the client and the server. If you'd like to try out this application or see exactly how it is written, you can find the client in the file *apps/electric*, and the server in the file *apps/elserver*, in the source code that accompanies this book.

7.8.1 Downloading Appointments from the Server

When each client starts up, it opens a socket to the server and sends the following message:

```
notify $env(LOGNAME) {
    receive {%date} {%time} {%alarm} {%comments}
} {
    startup
}
```

This tells the server that the client is interested in the appointments for a particular user. In this case, we're taking the user's name from the environment variable `LOGNAME`. On Unix systems, this variable contains the current login name. On other systems, this variable may not be set, so we'll also provide a way to set the user name in a preferences file.

The server reads the `notify` command and executes it in a safe interpreter. This causes the server to send back appointment data using two response templates, as shown in Figure 7.14. The first template is returned for each of the user's appointments, with data substituted into the following fields:

`%date`	appointment date (*`mm/dd/yyyy`*)
`%time`	appointment time (`8:00am`, `8:30am`, *etc.*)
`%alarm`	non-zero means that a reminder alarm is set

`%comments`	description for this appointment
`%user`	user that owns this appointment

So in this example, the server returns a series of `receive` commands, like this:

```
receive {03/10/1997} {9:00am} {1} {design review}
receive {03/25/1997} {1:00pm} {1} {conference call}
  ...
receive {07/15/1997} {8:00am} {1} {Tcl/Tk Workshop '97}
```

The client reads each `receive` command and executes it in its own safe interpreter. This causes the client to store the data for each appointment in its memory.

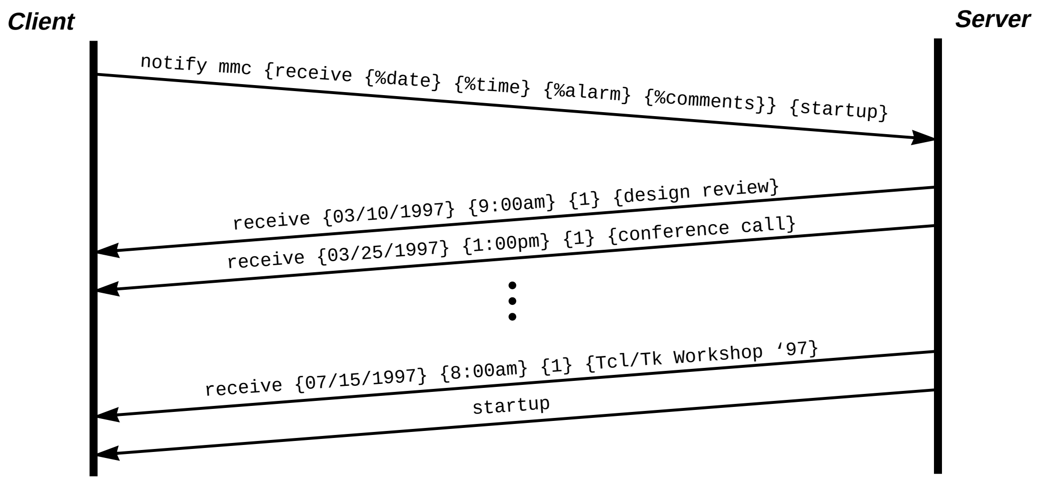

Client
Server
notify mmc {receive {%date} {%time} {%alarm} {%comments}} {startup}
receive {03/10/1997} {9:00am} {1} {design review}
receive {03/25/1997} {1:00pm} {1} {conference call}
receive {07/15/1997} {8:00am} {1} {Tcl/Tk Workshop '97}
startup

Figure 7.14. The *Electric Secretary* client sends a `notify` message to the server and gets back appointment data.

The second template is returned after all of the appointments have been sent. This lets the client know that the download is complete. Downloading all of the appointment data may take a minute or two, so while the client starts up, it will display an animated placard, letting the user know that the program is active. We'll see how the placard is implemented in Section 8.1.3.

When the download is complete, the client receives a `startup` command. This triggers an alias in the safe interpreter:

```
set clientParser [interp create -safe]
$clientParser alias startup placard_destroy
...
```

As you can see, the `startup` command invokes a procedure called `placard_destroy`, which takes down the placard and brings up the main application window.

7.8.2 Sending an Appointment to the Server

Once the client has started up, you can select any date on the calendar and view the appointments for that date. You can modify any appointment by typing your comments right into the appointment window. When the client notices that you have changed an appointment, it sends an `appointment` command to the server, like this:

```
appointment mmc 02/14/97 8:00am 0 {Valentine's Day}
```

The server reads this command and executes it in a safe interpreter. This causes the server to do two things:

- It adds the appointment to its own database.
- It sends the appointment to all other clients who are interested in this user, as shown in Figure 7.15. How does the server know which clients are interested? Remember, each client sends a `notify` command when it starts up, registering its interest in a particular user. The server not only sends the appointments for that user, but it also makes a note of the client's interest, and it saves the client's response template. When anything changes for that user later on, it notifies the client. With this feature, you can add an appointment to someone else's calendar, and it will instantly appear on their screen!

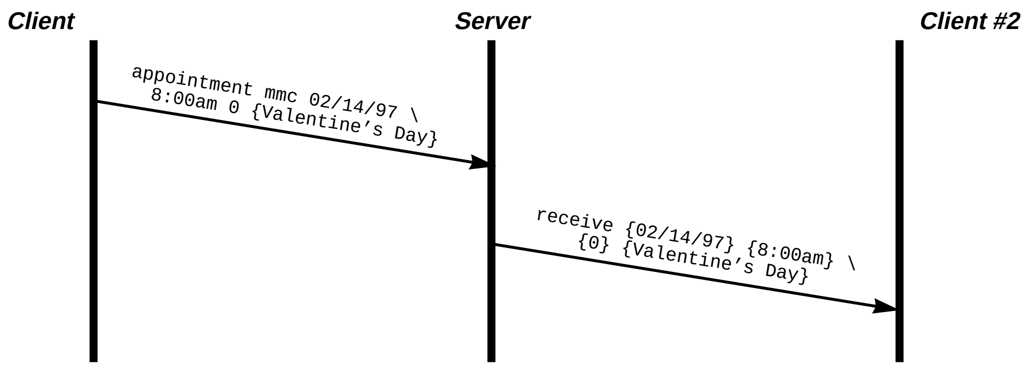

Client
Server
Client #2
appointment mmc 02/14/97 \
8:00am 0 {Valentine's Day}
receive {02/14/97} {8:00am} \
{0} {Valentine's Day}

Figure 7.15. One client sends an appointment to the server, and the server automatically notifies another client.

7.8.3 Handling Schedule Conflicts

There are lots of calendar programs that let you store appointments. But the *Electric Secretary* does more than that. It manages the calendars for many different people, so if you want to arrange a meeting, it can help you find times when everyone is free. And when you set a meeting date, it will automatically broadcast the change to everyone's screen.

For example, suppose you want to arrange a staff meeting for the junior executives. First, you bring up the *Preferences* dialog and create a group called `jrstaff`, as shown in Figure 7.16(a). This will add an entry for *jrstaff* onto the *Consult* menu. Next, you select a particular date for the meeting. Now, if you select the *jrstaff* entry on the *Consult* menu, you'll get a composite schedule for that day, as shown in Figure 7.16(b). Conflicting times will be marked in red, showing the names of the people that are busy. If you can find an empty slot, you can enter a new appointment. Pressing *Update* will add the appointment to your own calendar, but pressing *Update All* will add the appointment to the calendars of everyone in the group! If the group members happen to be looking at their calendars, they'll see the change immediately.

Let's see how this works. When you select a group from the *Consult* menu, the client sends the server a `consult` message, which looks something like this:

```
consult 03/10/1997 {allegra alex max katie} {conflicts %users}
```

The first two arguments tell the server the desired date and the list of users. The server checks the schedules for these users and builds a list of time conflicts. Then it substitutes the information into the `%users` field of the response template, and sends the client the response, which in this case looks something like:

```
conflicts 8:00am allegra 8:30am {allegra alex} 9:00am {allegra max} ...
```

When the client receives the `conflicts` command, it pops up a dialog with an appointment editor, showing the conflicts.

When you press the *Update* button, the client sends new appointments off to the server, using the `appointment` command that we saw earlier. For example,

```
appointment max 03/10/1997 11:00am 1 {Junior Staff Weekly Meeting}
```

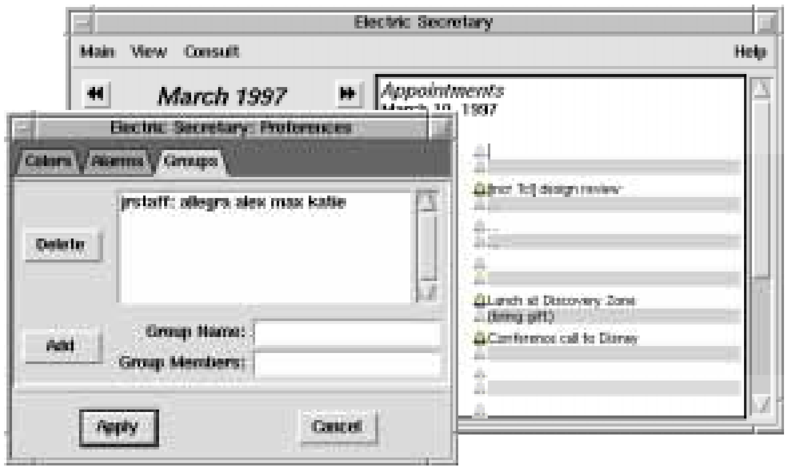
Similarly, when you press the *Update All* button, the client sends a series of appointments to the server, one for each user in the group:

```
appointment allegra 03/10/1997 11:00am 1 {Junior Staff Weekly
Meeting}
appointment alex 03/10/1997 11:00am 1 {Junior Staff Weekly Meeting}
appointment max 03/10/1997 11:00am 1 {Junior Staff Weekly Meeting}
appointment katie 03/10/1997 11:00am 1 {Junior Staff Weekly Meeting}
```

7.8.4 Preferences

The *Preferences* dialog shown in Figure 7.16(a) lets you customize the colors, alarm characteristics, and work groups for the *Electric Secretary.* When you make any changes, your new settings are saved in a file called *.esecrc* in your home directory. Each time you start up the client application, it loads your settings from this file.

(a)



jrstaff: allegra alex max katie

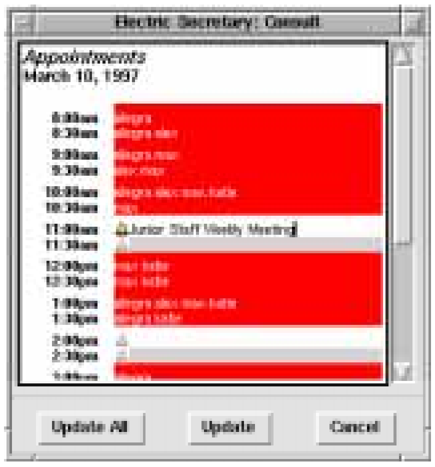
Group Name:

Group Members:

Appointments
March 10, 1997

[incr Tcl] design review
Lunch at Discovery Zone
(bring gift)
Conference call to Disney

(b)



Appointments
March 10, 1997

```
 8:00am  allegra
 8:30am  allegra alex
 9:00am  allegra max
 9:30am  alex max
10:00am  allegra alex max katie
10:30am  katie
11:00am  Junior Staff Weekly Meeting
11:30am
12:00pm  max katie
12:30pm  max katie
 1:00pm  allegra alex max katie
 1:30pm  allegra katie
 2:00pm
 2:30pm
```

Figure 7.16. (a) The *Preferences* dialog lets you define groups of users. (b) When you consult the schedule for a group, conflicts are marked in red.

We could make up any format for the preferences file, but following the advice in Section 7.6.5, we'll invent some Tcl commands. A typical preferences file might look like this:

```
# Electric Secretary Preferences
# updated: Mon Mar 10 01:09:23 EDT 1997
user mmc
colors white black red
alarms {All of the above} {Pop-up reminder}
groups {{jrstaff: allegra alex max katie}}
```

Four commands are used to convey the preference values. The `user` command sets the user name, which is stored in the `LOGNAME` environment variable. The `colors` command sets the background, foreground and selection colors used in the calendar. The `alarms` command sets the alarm characteristics. You see, each appointment has an alarm bell that can be turned on or off. If it is turned on, the client program will warn you when the time for that appointment draws near. The client can ring a bell, pop up a reminder notice, or do both, depending on the settings in the `alarms` command. Finally, the `groups` command specifies the user groups in the *Consult* menu that let you check for scheduling conflicts.

We can load the preferences file by executing it in a safe interpreter, like this:

```
set prefsParser [interp create -safe]
$prefsParser alias user esec_prefs_cmd_user
$prefsParser alias colors esec_prefs_cmd_colors
$prefsParser alias alarms esec_prefs_cmd_alarms
$prefsParser alias groups esec_prefs_cmd_groups
set cmd {
    set fid [open [file join $env(HOME) .esecrc] r]
    set script [read $fid]
    close $fid
    $prefsParser eval $script
}
if {[catch $cmd err] != 0} {
    notice_show "Error in preferences file .esecrc:\n$err" error
}
```

If anything goes wrong, we pop up a notice dialog to display the error.

We use aliases to link the four preference commands to the procedures in the main interpreter that store the settings. For example, the `user` command is aliased to `esec_prefs_cmd_user`, which is implemented like this:

```
proc esec_prefs_cmd_user {name} {
    global env
    set env(LOGNAME) $name
}
```

It simply stores the user name in the `LOGNAME` environment variable. The other preference procedures are implemented in a similar manner.

7.8.5 Persistent Storage

When the server receives an appointment, it adds that appointment to persistent storage. That way, if the server crashes or its host machine goes down, the appointment data won't be lost. When the server restarts, it can find all of its data on disk.

We could have used a relational database to store the information. There are a few different extensions that you can add to Tcl to access commercial database packages. For example, you can use *Oratcl* to access Oracle databases, and *Sybtcl* to access Sybase databases.[13]

But for this simple application, we'll use a plain ASCII file. When our server receives an appointment, it'll echo an `appointment` command to the storage file. Over time, the storage file will build up a complete history of appointments. It might look something like this:

```
appointment boss 02/21/1997 8:00am 1 {Executive project review:
explain why project is behind schedule}
appointment allegra 03/10/1997 8:00am 1 {Finish Tcl/CORBA interface}
appointment alex 03/10/1997 1:00pm 1 {Call John Ousterhout}
appointment max 03/10/1997 9:00am 1 {[incr Tcl] design review}
appointment katie 03/10/1997 9:00am 1 {[incr Tcl] design review}
```

When the server starts up, it can restore all of its data by executing this file in a safe interpreter. It just so happens that we have a safe interpreter with an `appointment` command—we're using it to filter the client requests. So we can use this same interpreter to load data from persistent storage.

7.8.6 Conclusions

Using a networked, client/server architecture has two advantages:

- It centralizes our data storage.
- It lets clients communicate and share information.

All of the messaging in this application is based on Tcl commands. The client and server programs communicate by exchanging Tcl commands. The client stores its preferences as Tcl commands. And the server handles persistent storage through Tcl commands. Tcl provides a powerful way of expressing information. As long as we're careful to execute these commands in a safe interpreter, we can keep the power of Tcl safely under control.

Client/server programs don't have to be complicated. As you can see, our server is just 200 lines long, and it has a small vocabulary. It handles three commands: `notify`,

13. You can download these extensions from:
http://www.NeoSoft.com/tcl/ftparchive/sorted/databases

`appointment` and `conflicts`. But when these simple commands are used together, they form a powerful application.