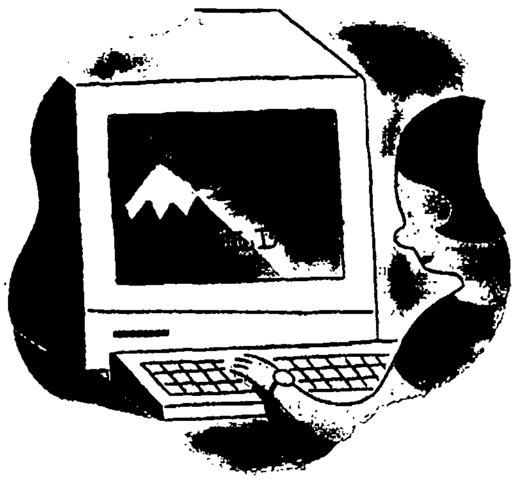
Who Lives Near Here?

by Jeffreys Copeland and Haemer

For the past few months, we've promised to attack the problem of how to find "everyone in my address book who lives *near* Boulder, CO."



Why "near"? Because that's usually what you want. For example, when we go to L.A., we don't really know anyone who lives *in* L.A.–does anybody?–but we do know folks in Santa Monica and Pasadena.

(This month's Jeff and Jeff trivia: Both of us lived in Pasadena for a few years in our dim past, though a decade apart. When we met in 1984, it took us no more than five minutes to discover that we'd lived in the same house.)

We've been stalling on giving you our solution for the least noble of reasons: Unlike printing envelopes or maintaining address books or any of the other problems we've shown you how to solve in this series, this is a problem we wanted to solve but, up until now, hadn't.

Starting this month, we'll attack this problem head-on. If you like our approach, steal it. If you have a better approach, let us know.

Naturally, before we start, we'll show our solutions to some exercises we left for you in our last installment.

Last time, we showed how to use `trap` to provide special handling in a shell script for signals 0, 1, 2, 3 and

Jeffrey Copeland (`copeland@alumni.caltech.edu`) *is a member of the technical staff at QMS's languages group, in Boulder, CO. His recent adventures include internationalizing a large sales and manufacturing system and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization, typesetting, cats and children.*

Jeffrey S. Haemer (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

Figure 1

```
 1) SIGHUP     2) SIGINT      3) SIGQUIT    4) SIGILL
 5) SIGTRAP    6) SIGIOT      7) SIGEMT     8) SIGFPE
 9) SIGKILL   10) SIGBUS     11) SIGSEGV   12) SIGSYS
13) SIGPIPE   14) SIGALRM    15) SIGTERM   16) SIGURG
17) SIGSTOP   18) SIGTSTP    19) SIGCONT   20) SIGCHLD
21) SIGTTIN   22) SIGTTOU    23) SIGIO     24) SIGXCPU
25) SIGXFSZ   26) SIGVTALRM  27) SIGPROF   28) SIGWINCH
29) SIGUSR1   30) SIGUSR2
```

15. Then, we asked you why we had chosen these.

Although `trap` is common to every POSIX shell, each UNIX system has its own idiosyncratic suite of signal names and numbers. Fortunately, POSIX.2 gives us a command to get the map of signal names to signal numbers. On one of our systems, Figure 1 shows what we get when we ask `$ kill -l`.

The names are mnemonic; for example, `SIGFPE` is the signal generated by a floating-point exception and is, in many cases, traditional.

The first three signals–hangup, interrupt and quit–are what you get when you press the keys `^D`, `^C` and `^\`. The 15th signal, terminate, is the default signal sent by `kill` with no signal number. The unlisted "signal," `0`, is what the script sees when the shell terminates normally. These are all pretty routine ways for a script to die, and we trapped them so that the script could clean up temporary files before exiting.

The names are more portable than either the numbers or the key bindings. POSIX guarantees that some of the names (including those listed above) will be supported, and users can use `stty` to set their key bindings.

Of course, not every choice is good. For example, typing `$ stty intr x` makes it impossible to type `$ exit` and typing `$ stty intr y` makes it hard to change the binding back to anything else.

The IBM-proprietary, multibyte character set found on early versions of AIX showed the other side of this coin. This benighted character set stole some of the bytes used by traditional UNIX systems to generate signals from the keyboard and turned them into bytes that signaled the start of a multibyte character. It was impossible to type `^\` to abort a process because whenever the user typed `^\`, the terminal driver would just sit patiently, awaiting the next byte of the two-byte character that it thought was coming.

One really nice thing POSIX.2 added was the ability to specify signals by name. We can now say `kill -SIGHUP` in our scripts, instead of `kill -1`, and

```
trap "rm -f $TMPFILE" EXIT HUP INT QUIT TERM
```

instead of

```
trap "rm -f $TMPFILE" 0 1 2 3 15
```

The other question we dropped in your lap last time was how we might have generated our timing data. This question is full of surprises. Two things that surprise us about it are how often we ask it and how often other

Figure 2

```
#!/usr/local/bin/perl -w
# $Id: seq.pl.scr,v 1.1 1995/07/11 17:40:21 jeff Exp $

$USAGE = "usage: $0 n1 [n2]";

sub seq {           # generate a sequence of numbers
                    # seq(4,6) generates the array (4,5,6)
                    # seq(5) generates (1,2,3,4,5)

        if (@_ == 1) {
              @_ = (1, @_);
        }
        local ($low, $high) = @_;
        return ($low..$high);
}

@ARGV || die $USAGE;

 if ($ARGV[1] && ($ARGV[0] > $ARGV[1]))  { # seq 6 3 generates 6 5 4 3
        @ARGV = ($ARGV[1], $ARGV[0]);
        $reverse++;
}

@foo = seq(@ARGV);
@foo = reverse(@foo) if $reverse;
print "@foo\n";
```

folks don't.

POSIX.2 and therefore AIX provide the traditional UNIX time routine, which gives the time that a command takes to complete. This is the right building block, but you need to know how to use it.

First, the command writes three numbers to standard error: real (clock) time, system time and user time. Here's an example:

```
$ time ls /bin > /dev/null
        0.02 real     0.00 user     0.02 sys
```

The clock time depends on a lot of things, like how many other processes are on the system and whether or not the program is already in memory. Ignore it. The others are the amount of time the code spends in and out of the kernel, respectively. For a first cut, just add them.

Why isn't that enough? In our experience, different runs give somewhat different answers.

```
$ time find /usr/lib -type f -print > /dev/null
      0.16 real      0.02 user     0.04 sys
$ time find /usr/lib -type f -print > /dev/null
      0.04 real      0.01 user     0.03 sys
$ time find /usr/lib -type f -print > /dev/null
      0.04 real      0.01 user     0.03 sys
$ time find /usr/lib -type f -print > /dev/null
      0.04 real      0.00 user     0.03 sys
$ time find /usr/lib -type f -print > /dev/null
      0.03 real      0.00 user     0.04 sys
```

As you can see, these times span a factor of two. We could average them, but the arithmetic mean isn't very robust, especially if we don't have a reason to believe that the underlying statistical distribution is symmetrical and relatively well-behaved. A better idea is to take the median–the middle value of several trials. Here's our code:

```
#!/bin/sh
PATH=/bin:/usr/bin:/usr/local/bin

USAGE="usage:$(basename$0)command[arguments]"
abort() {
        echo $* 1>&2
        exit 1
}
test $# -ge 1 || abort $USAGE

for i in $(seq 1 11)
do
        time $* > /dev/null
done 2>&1 |
awk '{print $3 + $5}' |
median
```

Looking at this code, we see some pieces that are interesting and useful in their own right: seq generates a sequence of numbers, and median calculates medians.

We often use seq in loops of various flavors. We stole the name from Kernighan and Pike's *The UNIX Programming Environment* (Prentice-Hall Inc., 1984, ISBN 0-13-937699-2 or 0-13-937681-X), a book you should own. Figure 2 shows our version.

We call the second program median because we usually use it for calculating the median of its input. However, if median is called with a first argument between 0 and 100, it calculates the percentile. For example, median 75 gives the upper quartile. We've used it for everything from building the timing command shown earlier to browsing the California Achievement Test scores provided by our local school district. Figure 3 shows our code.

The simplicity of the code is noteworthy. We wrote the program in Perl because we can call on Perl's built-in sort command. We could also have used the sort command itself, in a shell script, but we wanted to do floating-point arithmetic.

Onward and Upward with Geography

Back to the question we started with: How do we find folks living "near" a particular location (whatever "near" may mean)?

We've used the *median* program for everything from building our timing command to browsing the California Achievement Test scores provided by our school district.

Last month, we observed that solving this problem required a geographical information system (GIS), but that the proliferation of interesting Net sites might make it possible to avoid having to spend money and disk space to store the data on our own machines. For example, we could look for our information on a GIS provided by the University of Michigan:

```
$ telnet martini.eecs.umich.edu 3000

Trying 141.212.99.9...
Connected to martini.eecs.umich.edu.
Escape character is '^]'.
# Geographic Name Server, Copyright 1992
# Regents of the University of Michigan.
# Version 8/19/92. Use "help" or "?" for
# assistance, "info" for hints.
```

```
boulder, co
0 Boulder 1
08013 Boulder
2 CO Colorado
3 US United States
R county seat
F 45 Populated place
L 40 00 54 N 105 16 12 W
P 76685
E 5344
Z 80301 80302 80303 80304 80306
z 80307 80308 80309 80310 80314
Z 80322 80323 80328 80329
.
quit
Connection closed by foreign host.
```

The lines marked Z give all the ZIP codes in Boulder, the line marked 1 gives the county name and the line

Don Libes, a hero of the software revolution, has given us Expect, a general-purpose scripting language for handling fundamentally interactive tasks.

marked L gives Boulder's latitude and longitude. This is something around which we might build a fine solution.

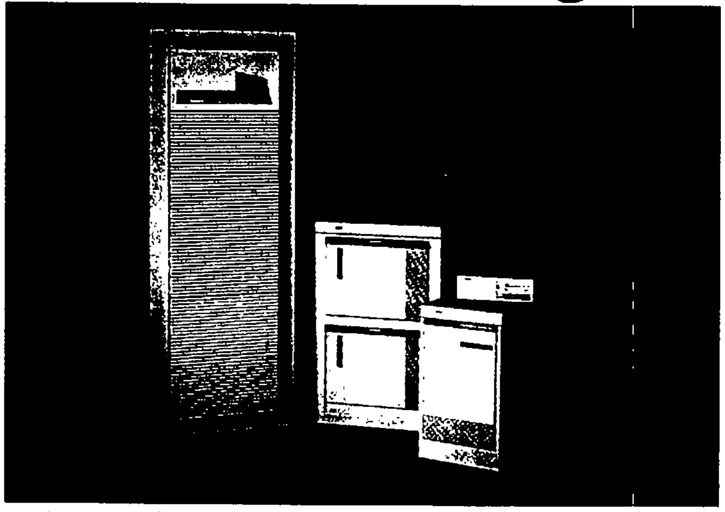
There are some catches, though. First, we can't search the database directly. We must use the interface and search engine provided by the remote host. Second, we can't easily capture the output for further analysis. This GIS is really designed for interactive use.

Don Libes, a hero of the software revolution, has given us just what we need here: Expect, a general-purpose scripting language for handling fundamentally interactive tasks.

Expect lets us write programs that pretend to be users and that can capture, parse and manipulate the output of the program (or programs) on the other end.

For this job, we'll begin by developing an expect script that requests information from our GIS

Not only do they remember,
they never forget!



As intermediate and long-term data storage technology advances, so does your need for a qualified, full service integrator. Since 1986, Computer Upgrade has been providing leading edge, optical and tape data storage solutions, named OmniLib, for a wide range of applications. For a memorable solution to your data storage and retrieval needs, call Computer Upgrade today for an OmniLib solution that will never forget!

- Multi-host support including: IBM, SUN, HP, SGI, & Alpha, etc.
- Optical and Tape storage systems.
- Complete UNIX jukebox management software with AMASS, and DataMgr File Migration HSM software from EMASS. Single mount point, magnetic disk drive emulation, single filesystem view, and network compatible.
- National and International installations and maintenance.
- From 20GB to 75 plus terabytes of storage, with 2 to 16 drives.

Computer Upgrade Corporation
1921 Sampson Avenue, Corona, CA 91719
(909) 278-5626 FAX (909) 278-9093, email – cuc@optical.com

AMASS authorized Distributor from EMASS

OmniLib is a trademark of Computer Upgrade Corporation.
All other trademarks are the property of their respective holders.

Circle No. 2 on Inquiry Card

and then captures it.

We'll start very slowly. Figure 4 shows a test program that does almost nothing.

This program sets up a `telnet` session with the local host (`telnet 0` should work whether or not your Net connection is up), scans "remote" input for a prompt (`set remote_prompt "login: "`, later followed by `expect $remote_prompt`), sends the user name when it sees the prompt (`{ send "$env(LOGNAME)" \r" }`) and then turns control back to the user (`interact`).

We won't explain the syntax in detail, but Expect is built on top of John Ousterhout's Tcl and uses its syntax and semantics.

There are at least two good books on Tcl–*Tcl and the Tk Toolkit* by John Ousterhout (Addison-Wesley, 1994, ISBN 0-201-63337-X), and *Practical Programming in Tcl and Tk* by Brent Welch (Prentice-Hall, 1995, ISBN 0-13-182007-9)–and one on Expect–*Exploring Expect* by Don Libes (O'Reilly & Associates Inc., ISBN 1-56592-090-2). If you're reluctant to buy three books when one is enough, Don's is self-contained.

Once this is working, let's try logging in to the remote host instead:

Figure 3

```
#!/usr/bin/perl
# $Id: median.pl.scr,v 1.1 1995/07/11 17:40:21 jeff Exp $

$USAGE = "usage: $0 [percentile] [filename]";

sub percentile {        # the percentile score
       # 1st arg a pointer to an array, 2nd, the percentile
       # e.g., percentile(50, @A) returns the median
       # (defaults to median without a second arg)

    local ($percentile,@A) = @_;
    $percentile /= 100;
    local ($fold, $ifold);

    @A = sort {$a <=> $b} @A;

    $fold = @A*$percentile;
    $ifold = int(@A*$percentile);
    if (($fold - $ifold) > .001) {
       return $A[$ifold];
    } else {
       return ($A[$ifold] + $A[$ifold-1]) / 2;
    }
}

$percentile = 50;          # default to median
if (@ARGV > 1) {
    $percentile = $ARGV[0]; shift;
}

die $USAGE if (($percentile < 0) || (100 < $percentile));

chop(@seq = <>);
print &percentile($percentile, @seq) . "\n";
```

```
#!/usr/local/bin/expect --

set remote_prompt "\n\.\r\n"

spawn telnet martini.eecs.umich.edu 3000
expect {
  $remote_prompt {send_user "Found prompt!\n"}
  default {send_user "Oops!\n"}
}
interact
```

In this second stage, we telnet to the remote machine and have the code scan for a different prompt–the GIS's lone dot (.). We also add a default action, which is taken if the remote host doesn't provide the GIS prompt within a reasonable amount of time (the default time is 10 seconds).

That's swell, but can we write code that does anything more interesting than log in? You bet. Take a look at this:

```
#!/usr/local/bin/expect --

set remote_prompt "\n\.\r\n"

spawn telnet martini.eecs.umich.edu 3000
expect {
       $remote_prompt {send "help\r"}
       default {
              send_user "Oops!\n"
              exit
       }
}

log_file telnet_log

expect {
       $remote_prompt {
              log_file
              send "quit"
       }
       default {send_user "Oops!\n"}
}
expect "Connection closed"
```

This script logs in, waits for the GIS prompt, sends the

Figure 4

```
#!/usr/local/bin/expect--
# begin log-in sequence onto the local host

set remote_prompt "login: "

spawn telnet 0
expect $remote_prompt { send "$env(LOGNAME)" \r" }
interact
```

command help, captures the GIS' response in the file telnet_log and then logs out cleanly as soon as it sees the telnet message

```
Connection closed by foreign host.
```

We're getting close!

We'll tie up this month's column with one last expect script. This one retrieves the GIS response to a keyword specified on the command line when the script is invoked:

```
#!/usr/local/bin/expect--

set remote_prompt "\n\.\r\n"
set keyword [lindex $argv 1]

spawn telnet martini.eecs.umich.edu 3000

expect {
   $remote_prompt {
      log_file telnet_log
      if {[string compare $keyword ""] == 0} {
         interact {
            "quit\n" {
               log_file
               send "quit\r"
               expect {"Connection closed"}
            }
         }
      } else {
         send "$keyword"
         expect $remote_prompt
         log_file
         send "quit\r"
         expect "Connection closed"
      }

   }
   default {send_user "Oops!\n"}
}
```

Next time we'll try building on this to construct interesting queries and analyze the data we get back.

"Where do I get Expect?" you're probably asking. Simple! Off the Net. The latest version is always available in pub/expect at ftp.cme.nist.gov. The latest version of Tcl, which you'll need to build and run Expect, is in the directory pub/tcl on ftp.cs.berkeley.edu.

Exercising for the Next Installment

After months of working around the problem of geographic proximity, we're about to provide a solution. By way of a massive exercise for the reader, how would you go about determining who lives near who in your address book, given the tools we've developed to date?

What does geographic proximity mean anyway? If you live in Moscow, is St. Petersburg nearby? How about after you've traveled from Houston to Moscow?

How would you use the latitude and longitude in concert with the information in your address book? Can ZIP code be used as a proximity metric? If telephone area codes were consistently available in the Michigan GIS, how would you use those?

As a larger *gedanken* exercise: What does geographic proximity mean anyway? If you live in Moscow, is St. Petersburg nearby? How about after you've traveled from Houston to Moscow? Is St. Petersburg relatively closer to Moscow? How can we tune our tool to vary the definition of "nearby"?

We believe in the "99 bottles of beer" test for programming languages. We try to write a program in each new language we learn that prints all the verses to the familiar junior high school bus trip song. As a more amusing exercise, write the Expect version. (We'll warn you: Don Libes has written a version already–and a better one than any we've written. No fair peeking at the examples in the Expect distribution to find it.)

See you next time! ▲