Let's Synchronize Our Watches

by Jeffreys Copeland and Haemer

10...9...8...7...6...5...4... 3...2...1...Happy New Year!

How did you know when it was midnight? Really midnight? Nerds like us might have called 303-499-7111, and spent the evening listening to the National Institute of Standards and Technology's (NIST, nee, National Bureau of Standards) time signal as broadcast on WWV.

(Cinematic aside: Which movie is your New Year's Eve favorite? We are agreed on "It's a Wonderful Life" as the Christmas movie of choice, but have been debating the merits of "The Fabulous Baker Boys" versus "The Apartment" for the best New Year's Eve flick.)

But we digress. There's an easier way to find out the time than sitting with the telephone pressed to your ear all night, and that's to resync your computer's clock directly to someone who knows the right time, like NIST.

Your Friend, telnet

The Internet utility `telnet` dates back more than 20 years. Usually, it's used to connect one computer to another remotely. Because it doesn't require tight communications on both ends, it works better over long distances than `rlogin`.

However, logging into a remote computer isn't the only trick that `telnet` knows. If you take a look at `/etc/services`, you'll find a list of functions that can be accomplished

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.*

across a TCP or UDP link. One of `telnet`'s tricks is to allow you access to any of those functions.

For our first experiment, pick a machine close by–for argument's sake, we'll dub it *barney*–and utter,

```
telnet barney echo
```

to your computer. You'll find yourself connected to *barney*, and have it repeat back everything you type. Next, try,

```
telnet barney daytime
```

You'll be rewarded with *barney*'s notion of the correct date and time. Quick, type `date` and see if your clock agrees with *barney*'s. It almost certainly doesn't.

OK, let's head off a bunch of mail by admitting that we know there are standard utilities for solving this. We use `rdate` on our Sun desktop machines, for example.

The *daytime* service gives us the local time on the remote machine as text, without the time zone, while the *time* service provides a UNIX *time_t* word in binary form.

Ultimately, the machines in our local cluster all synchronize their clocks to QMS's firewall, which synchronizes to the clock at `pogo.udel.edu`, which, in turn, relies on an atomic clock for its time.

Perl 5 for Making Remote Links

Perl 5 provides a number of features for connecting to a remote computer. We're going to use a variety of them. Basically, we'll establish a TCP connection with the remote machine, set up a socket to communicate with it, read the time from the remote computer and set our time to be the same.

Let's begin with some setup:

```
require 5.002;
use strict;
use POSIX;
# debug flag
my $d = $main::d ? $main::d : 0;
```

This ensures that we're executing at least Version 5.002 of Perl, that we strictly enforce safety, that we include the POSIX.1 interfaces and that we have an alias for the debug flag. The latter is necessary because we are using `strict`–thus, we must fully qualify the variable.

Having done the setup, we'll proceed from the bottom up and write the service routines first, beginning with one to set up the connection to the remote computer. For example,

```
sub establish_connection {
  use socket;

  # Declare variables to keep
  #    'use strict' from whining.
  #    Here, we declare them all
  #    at the top of the routine.

  my $s_host = shift;
  my($c_host, $c_socket, $c_addr);
  my($s_addr, $s_socket, $port);

  my $pnum;
  my $pname = 'tcp';
  my $stype = SOCK_STREAM;
```

We use `socket` to include the interfaces for interprocess communications. Then, we declare all the variables we need, beginning by collecting the `socket` host, `$s_host`, from the argument list. We finish this by declaring the protocol and socket type we'll use.

Next, we'll get the protocol number from `/etc/services` and establish the `socket`.

```
  # Make the socket filehandle.
  $pnum = getprotobyname $pname;
  socket(S, PF_INET, $stype, $pnum)
     or die "socket: $!";
  warn "socket ok" if $d;
```

Note: `getprotobyname()` is not the best name for a function call; it's too confusing. Do we mean to get the prototype, or the proto-what? A better name would have spelled it out as `getprotocolbyname()`.

Following that, we set up the `socket` and its address, bind the `socket` to handle `S`, and issue more status messages if we have debugging turned on.

```
  # Give our socket an address.
  $c_host = (POSIX::uname)[1];
  $c_addr   = inet_aton $c_host
     or die "no address: $c_host";
  $c_socket = sockaddr_in(0, $c_addr);
     # 0 means let kernel pick
  bind(S, $c_socket) or die "bind: $!";
  warn "bind ok" if $d;
```

Last, we establish the connection with the other computer, and read not the `daytime` service but the `time` service. The difference is that the `daytime` service gives us the local time on the remote machine as text, without the time zone, while the `time` service provides a UNIX `time_t` word in binary form. So, we continue with

```
  # Call up the server.
  $port = getservbyname('time', $pname)
     or die "No port";
  $s_addr   = inet_aton $s_host
     or die "no address: $s_host";
  $s_socket = sockaddr_in($port, $s_addr);
  connect(S, $s_socket) or die "connect: $!";
  warn "connect ok" if $d;
}
```

Now that we've got a routine to do the communications, we need a way to set the time on our local machine to match the time on the remote machine.

```
sub synchtime {
  # Here, we declare variables as they're used.

  my $rtime = shift;
  my $SECS_of_70_YEARS = 2208988800;
  #  From 1900 to the Epoch
  #  70*365*24*60*60 (70 regular years)
  #  + 17*24*60*60 (17 leap-days)

  my $histime = unpack("N", $rtime)
      - $SECS_of_70_YEARS ;
     # "N" is network-order long
  my $settime = POSIX::ctime $histime;
  chomp($settime);
```

We are assuming that we take `time_t` as an argument, and our task is to decode that word. We now need to know how many seconds in 70 years to enable us to correct the clock for the UNIX epoch, January 1, 1970.

Next, we use the Perl `unpack()` function to expand the word we got across the network and, after correcting for the epoch, we use that value to get the ASCII rendition of the current time. We finish by removing the trailing new lines from `$settime`.

The `chomp` operator is a slightly safer version of the older Perl `chop` operator, because it ensures the character it's chopping is actually a new line.

What good does it do us to have the remote date in text format? Well, it's a convenient form to feed to the GNU `date` utility. Why use the GNU version? Because every version of `date` we looked at requires its arguments for the `-s` flag (to set the date) to be in a slightly different form. Some have seconds separated by a dot, some have the year preceded by the century, some prohibit inclding two digits for the century: 1997 becomes 97, for example. POSIX.2 specifies the format specifiers for outputting the date, but not the format for *inputting* it. We use the common, and widely available, GNU version to provide a *lingua franca* date format. Thus,

```
   # assume GNU 'date'
  my $cmd =
   "/usr/local/bin/date -s '$settime'";
  warn $cmd if $d;
  my $rc = system($cmd);
  warn "system($cmd) failed" if ($rc != 0);
}
```

By the way, because GNU `date` allows you to set the date in nearly any format, it has the interesting feature of being a date format translator. For example, if we say

```
date +"%D" --date "31 Oct 1996"
```

we get `10/31/96` back. Even better, if we say

```
date --date "a week ago"
```

`date` responds `10/22/96`. This improves the usefulness of the utility by a large measure.

This leaves us with the main program to write. Given the subroutines we've already written, it's pretty short:

```
my $s_host = shift || 'localhost';

establish_connection $s_host;
chomp($_ = <S>);
warn $_ if $d;
synchtime $_;
close (S) or die "close: $!";
exit 0;
```

We begin by getting the target off the command line. Next, we use our first subroutine to get the connection to the remote host, providing status if we have debugging turned on. Last, we set the local time and finish.

We've always been fascinated by things like self-replicating programs. So the notion of including the documentation in the program itself is a natural trick for us. We can do this in Perl because there are some commands that are valid in both `troff` and Perl.

For example, we begin our script with the lines:

```
'di';
'ig 00 ';
#
```

```
# $Id: synchtime,v 1.11 1996/10/09 ....
```

The `di` and `ig` lines are simple string declarations to Perl, and tell `troff` to divert and ignore the text until a line containing `.00` appears.

Then, we finish the Perl code with the lines:

```
exit 0;
.00 ;
'di
.nr n1 0-1
.nr % 0
'; __END__
```

The first two lines exit from the Perl program, and then complete the lines that `troff` is ignoring. We begin a new Perl string declaration that contains the `troff` directives to close the diversion, and pretend that we're starting the first page of the man page again.

Finally, we declare the end of the Perl text and the beginning of the documentation text. This trick works in large part because a delimiter for Perl strings is also one of the two valid characters for starting a `troff` directive.

We also occasionally use a similar technique in our C code, and bodily include the man page surrounded by `#ifdef DOC` and `#endif`. As we've discussed in this column before, we still disagree about the utility of Don Knuth's literate programming technique.

In any event, the man page text we include is

SYNCHTIME(1)

NAME

synchtime–synchronize the system clock with another machine on the Net.

SYNOPSIS

`synchtime [-d] hostname`

DESCRIPTION

Synchtime uses the network 'time' service (port 37) to get the time from the named host, then calls GNU's `date -s` to set the time on the local machine.

FILES

`/etc/hosts`, `/etc/services`

AUTHOR

Jeffrey S. Haemer

SEE ALSO

`date`

DIAGNOSTICS

The `-d` flag produces some debugging information.

Whines if you don't run it as root, but doesn't do any damage.

BUGS

Must be run as root.

Requires GNU `date` because of the wild variety of formats that various vendors use for setting dates. Unfortunately, POSIX.2 specifies formats for getting dates but not for setting them. Luckily, GNU `date` is both widely available and has the best format going. Fortunately, GNU's `date -s` will let you give it almost any reasonable, verbose format, such as those returned by `ctime()`. It would be nice to use the "daytime" service (13), but the format returned doesn't give a time zone.

More Follies in Time Libraries

An anonymous correspondent points out that our chums at Sun Microsystems goofed in SunOS 4.1 and its successors. SunOS doesn't provide the standard ANSI C `mktime()` routine to convert a `struct tm` into a `time_t` value. Because `mktime()` hadn't been standardized by the time of its release, SunOS provides similar functionality with `timelocal()`. In principle,

```
mktime(localtime(time(NULL)))
```

should be the same as

```
time(NULL).
```

Unfortunately, the Sun routine ignores the `tm_isdst` flag in the `struct tm`, so during the summer, time isn't exactly invertible using `timelocal()`.

The other day, a copy of the November 1996 *Scientific American* came across our desk. In this issue, Ian Stewart devotes his Mathematical Recreations column to calendar calculations and points out that the book *Calendric Calculations*, by Dershowitz and Reingold, has been published by Cambridge University Press, ISBN 0-521-56474-3. We've already ordered our copy.

Stewart also provides a pointer to Reingold's Web page at `http://emr.cs.uiuc.edu/~reingold/calendars.html`, which features excerpts and sample code from the book.

That's all for now. Next month, we'll be looking at HTML again. This time, we'll discuss CGI techniques, and show you a bit of software we wrote for our eldest daughter's geography drills–and how we extended it to count electoral votes for the recent U.S. presidential election. If history is any guide, this will generate a lot of correspondence.

We ended up writing this month's column twice. We committed the sort of mistake that two professional nerds who have been at this for close to half a century shouldn't have made. One of us had a brain-skip and typed `rm 22.mm` instead of `rm 22.ps` as we were nearing completion. So, in the column following the HTML/CGI exploration, we'll discuss adding fail-safe protection to `rm`. Until then, happy trails. ▲