Monitoring room temperature

Within Bounds

In monitoring room temperatures with an affordable USB temperature sensor, the RRDtool open source library offers Holt-Winters forecasting to differentiate normal from abnormal deviations. *By Mike Schilli*

Do-it-yourself Linuxers trying out new USB devices have had to do some heavy lifting in the past. Serious tinkerers had to track down or even write their own drivers and embed them in the kernel code. The newest distributions make it a bit easier and more automatic. A TEMPer USB thermometer [1] recently purchased for $7 (shipping included) on eBay worked immediately without my even having to read the instructions (Figure 1).

When plugging the sensor into the PC's USB port, the kernel immediately recognizes it as a generic Human Interface Device (HID) and automatically applies the "raw" `hidraw` driver. Figure 2 shows the log entries for this event generated by the kernel in `/var/log/messages`.

The universal hidraw driver communicates with various devices, without having to know their characteristics. The bit manipulation that's needed for addressing the specifics of a USB temperature sensor in this case runs over a driver in userspace that the CPAN module Device::USB::PCSensor::HidTEMPer implements.

To read the temperature the sensor measures, Listing 1 first uses the `device()` method to find the device in the USB tree [2]. Because only one sensor is plugged in, only one device is listed; with multiple sensors, the `list_devices()` method would return a list of all devices detected. The internal sensor of the device gets addressed by the `internal()` method, and the `celsius()` call returns the temperature value as a floating point number with an accuracy of half a degree Celsius. If you're living in the US, you will most certainly be more familiar with temperatures measured in Fahrenheit, which the sensor driver supports as well via the `celsius()` method.

Behind the Scenes

A glance at the CPAN module source code makes it evident that behind the snazzy object-oriented API that easily returns the temperature values, quite a lot of low-level communication is going on: Control bit values are going back and forth, data buffers are created and emptied, and checksums are calculated. To identify the sensor, Linux reads its VendorID (1130) and ProductID (660C). That makes it inconsequential as to which USB port or hub the user plugs the device into. Under the hood, the CPAN module Device::USB (or rather its back end libusb C library) combs through the USB tree until it finds the device on the basis of the unique combination of vendor and product ID.

Meticulous Record Keeping

Now for something more useful than a simple temperature reader: The CPAN module App::Daemon creates a daemon process in Listing 2 that the admin can start and stop with `logtemp start` and `logtemp stop`. While running as a background process, it records the temperatures every minute and logs them in the `/var/log/temper.log` file (Figure 3). The Log4perl framework prepends a timestamp as well.

The command `sudo_me()` in line 18 is exported from the CPAN module Sysadm::Install and ensures that the script runs as superuser and, if that's not the case, restarts the script with a `sudo` call. Root privileges are required so the daemon can store the log data in `/var/log` and the PID of the newly created background process in `/var/run/temper.pid`. Immediately thereafter, the App::Daemon drops the root privileges

MIKE SCHILLI

Mike Schilli works as a software engineer with Yahoo! in Sunnyvale, California. He can be contacted at *mschilli@perlmeister.com*. Mike's homepage can be found at *http://perlmeister.com*.

Figure 1: The affordable TEMPer USB Thermometer sensor.

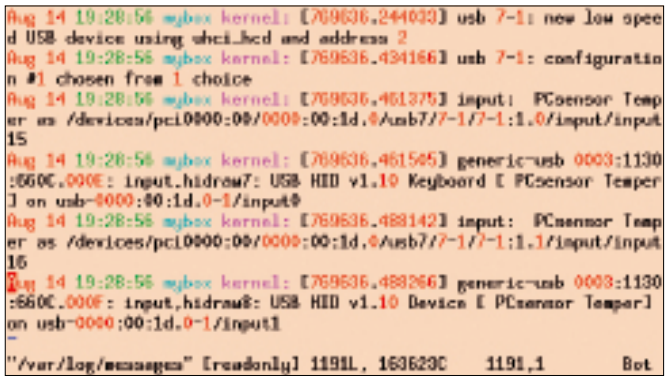

Figure 2: When the temperature sensor is plugged in, Ubuntu immediately recognizes the new USB device and assigns it the hidraw driver.

(for obvious security reasons) and continues as the user defined by the `$App::Daemon::as_user` variable. For this, the script extracts the user ID of the invoking user from the `SUDO_USER` environmental variable, courtesy of the `sudo` command calling the script. Again, if you want Fahrenheit values, change line 46 from `celsius()` to `fahrenheit()`.

The `daemonize()` request in line 30 has App::Daemon send the daemon into the background, which is visible to the calling user when the shell command-line prompt reappears only a split second after invoking the script. The user can then use the `tail -f /var/log/temper.log` command to follow along with the daemon (Figure 3). For testing purposes, the `logtemp -X` command option puts the process into the foreground, with log messages appearing on stderr.

Graphs Instead of Numbers

Measurement data in logfiles unfortunately rarely creates euphoria or leads to salary bumps, so it's no wonder graphical representation is the next step. The `rrdtool` toolset's `rrdgraph` program is especially useful in this context. If you can't stand its low-level syntax, the CPAN module RRDTool::OO provides the modern object-oriented one.

To convert the messages in the logfile, which are written in a human-readable time format (such as year/month/day hour:minute:second) to the one RRDtool customarily uses, Listing 3 uses the CPAN module DateTime::Format::Strptime and defines the desired date pattern in line 22. Line 23 sets the time zone to that of the local machine. The `while` loop starting in line 30 iterates through the logfile, and the regex pattern on line 31 extracts the lines with temperature entries and ignores others, such as start and stop messages.

Listing 3 stores the discovered measurement values in the array `@data_points` accompanied by their timestamps. At line 53 RRDtool goes to work by first defining a new round-robin database with enough to cover five months of measurements. To smooth out erratic values, it collects five-minute-interval measurement readings, as handled by the `step` value in line 54.

The RRDtool data collector uses its universal `GAUGE` data type for numerical values. The `for` loop starting on line 79 feeds the values and their timestamps (stored in the `@data_points` array) in the RRD database, using the `update` method. Calling the `graph` method on line 89 draws the graph (Figure 4). It also labels the axes and scales the values accordingly. Drawing a chart couldn't possibly be easier!

Watch for Disaster

The wavy nature of the graph naturally reflects the normal fluctuations in daily temperatures. However, to find out whether an exception occurred because of an unforeseen circumstance (the cat is lying on the sensor or the building is on fire), comparing the absolute values will not work because the daily ups and downs in temperature would trigger too many false positives.

RRDtool therefore provides a so-called "Aberrant Behavior Detection" feature that uses four parameters to determine "normal" temperature fluctuations and compares the measured values against them. It uses past measurements to prognosticate the future. If a certain

LISTING 1: celsius

```
01 #!/usr/bin/perl -w
02 ##############################
03 # celsius - Read TEMPer
04 #          sensor temperature
05 # Mike Schilli, 2010
06 # (m@perlmeister.com)
07 ##############################
08 use strict;
09 use local::lib;
10 use
11   Device::USB::PCSensor::HidTEMPer;
12
13 my $temper =
14   Device::USB::PCSensor::HidTEMPer
15   ->new();
16
17 my $sensor =
18   $temper->device();
19
20 if (
21  defined $sensor->internal())
22 {
23  print "Temperature: ",
24    $sensor->internal()
25    ->celsius(),
26    " C\n";
27 }
```

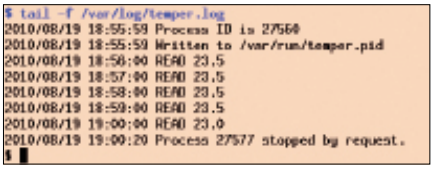

Figure 3: The thermo daemon records a temperature reading every minute in the logfile.

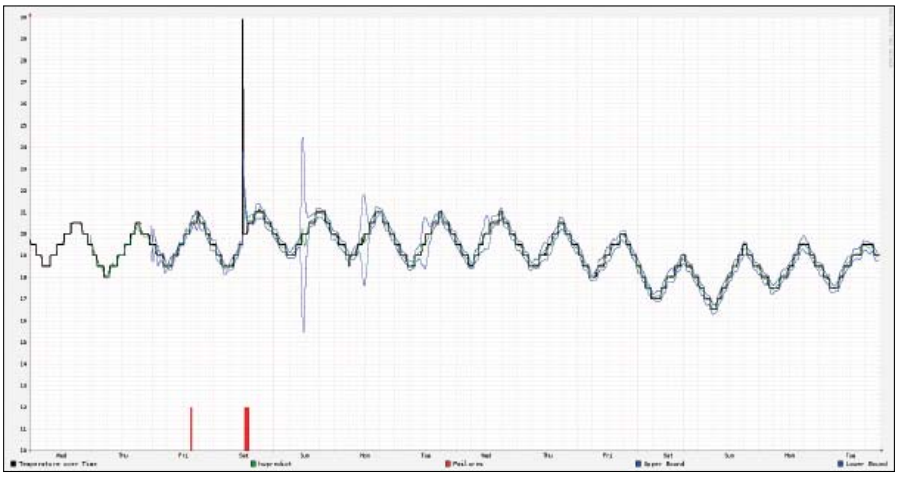

Figure 4: The graph (black) with Holt-Winters forecasting, the prognosis (green) and its confidence band (blue), and some resulting alarms (red).

number of predictions don't match reality within a given time frame based on the algorithm, the system returns errors, indicated in red (Figure 4).

Figure 4 shows the measurements in black, the prognosis in green, and the thresholds of what it considers "normal" values in blue. The alarms will be displayed as red lines at the bottom of the graph.

Unfortunately, the process does return some false alarms (for example, at noon of the third day), and it does not always pick up real errors reliably either. The administrator can always play with the four knobs until the results become satisfactory.

Of course, there is no guarantee that the next day's results won't create further alarms,, because adjusting the knobs is really more magic than science.

Turning Knobs

The admin can adjust the parameters `alpha`, `beta`, and `gamma` (each between 0 and 1) as well as the `seasonal_period` for the time period in which repeated temperature patterns are likely to occur, such as from day to day.

Small parameter values (close to 0) for `alpha`, `beta`, and `gamma` draw attention to events that can date back a bit, whereas values close to 1 base the prognosis on more recently recorded values. Whereas `alpha` controls the base values for the graph, `beta` works with its slope. `Gamma` determines the prognosis on the basis of recurrences in the predetermined `seasonal_period` intervals. Listing 3 sets `alpha=>0.1`, `beta=>0.0035`, `gamma=>0.5`, and `seasonal_period` to the number of data points in a day.

The system records an error if a number of values given in `threshold` falls outside the "confidence band" (i.e., the blue lines) during a time frame of the given `window_length`. RRDtool automatically determines the confidence band from the parameters provided, and there is no way to look under the hood or influence it directly.

A few eye-catching places occur in the graph. RRDtool doesn't offer any prognoses during the first two days because it needs a few cycles to determine the effect of the "seasonal component" on the prognosis. Curiously, the system expected the spike just before noon on the fourth day to be repeated at the same time the following days. Because it didn't happen, the system's expectations are lowered stepwise, until after a few cycles, expectations are back to normal.

What `rrdtool` does under the hood can be seen by turning on the Log4perl framework: It is sleeping within RRDTool::OO and waits until the user activates it. Figure 5 shows what goes on in RRDTool::OO, starting with the command to create the database, then selection of the update commands for logged temperature values, and finally the `graph` command that draws the actual graph.

RRDtool calls the Holt-Winters forecasting `HWPREDICT` and the expected sta-

LISTING 2: logtemp

```
01 #!/usr/bin/perl -w
02 ############################
03 # logtemp - Daemon logging
04 #           TEMPer readout
05 # Mike Schilli, 2010
06 # (m@perlmeister.com)
07 ############################
08 use strict;
09 use local::lib;
10 use
11   Device::USB::PCSensor::HidTEMPer;
12 use App::Daemon 0.10
13   qw(daemonize);
14 use Log::Log4perl qw(:easy);
15 use Sysadm::Install qw(:all);
16 use File::Basename;
17
18 sudo_me();
19
20 $ENV{SUDO_USER} ||=
21   "mschilli";
22
23 $App::Daemon::logfile =
24   "/var/log/temper.log";
25 $App::Daemon::pidfile =
26   "/var/run/temper.pid";
27 $App::Daemon::as_user =
28   $ENV{SUDO_USER};
29
30 daemonize();
31
32 while (1) {
33  my $temper =
34    Device::USB::PCSensor::HidTEMPer
35    ->new();
36
37  my $sensor =
38    $temper->device();
39
40  if (
41   defined $sensor->internal()
42    )
43  {
44   INFO "READ ",
45     $sensor->internal()
46     ->celsius();
47  } else {
48   ERROR
49     "No reading available";
50  }
51
52  sleep 60;
53 }
```

```
rrdtool 'create' 'data.rrd' '--start' '1282112701' '--step' '300'
 'DS:temp:GAUGE:600:U:U' 'RRA:AVERAGE:0.5:1:43200' 'RRA:HWPREDICT
:43200:0.1:0.0035:288:3' 'RRA:SEASONAL:288:0.5:2' 'RRA:DEVSEASONA
L:288:0.5:2' 'RRA:DEVPREDICT:43200:4' 'RRA:FAILURES:43200:14:18:9
'

rrdtool 'update' 'data.rrd' '1282112761:20'
rrdtool 'update' 'data.rrd' '1282112821:20'
...
rrdtool 'update' 'data.rrd' '1283322241:19'
rrdtool 'update' 'data.rrd' '1283322301:19'
rrdtool 'graph' 'bounds.png' '--width' '1680' '--height' '800' '-
-end' '1283322301' '--start' '1282112761' 'DEF:draw1=data.rrd:tem
p:AVERAGE' 'LINE2:draw1#000000:Temperature over Time' 'DEF:predic
t=data.rrd:temp:HWPREDICT' 'LINE1:predict#00FF00:hwpredict' 'DEF:
dev=data.rrd:temp:DEVPREDICT' 'DEF:failures=data.rrd:temp:FAILURE
S' 'TICK:failures#FF0000:.1:Failures' 'CDEF:draw5=predict,dev,2,*
,+' 'LINE1:draw5#0000FF:Upper Bound' 'CDEF:draw6=predict,dev,2,*,
-' 'LINE1:draw6#0000FF:Lower Bound'
"rrd.debug" 7 lines --71%--                    5,1           All
```

Figure 5: RRDtool commands for the graph in Figure 4 that the RRDTool::OO script generates.

```
==="usb", ==="664",
idProduct}=="600c", MODE="666"

"99-tempsensor.rules" 1L, 96C                 1,1           All
```

Figure 6: A new file in `/etc/udev/rules.d` directs the `udev` system to provision the USB device with `666` permissions, which allows anyone on the system to obtain temperature readings.

tistical deviation `DEVPREDICT`. Line 115 in Listing 3 defines the deviation with the alias `dev`, which lines 132 and 139 each use to draw the confidence band. In typical RRDtool RPM notation, `predict,dev,2,*,+` stands for algebraic `predict + 2 * dev`, because RRDtool allows for deviations above and below up to twice the `DEVPREDICT` value above and below `HWPREDICT`.

Installation

Because the required CPAN temperature sensor module is not part of the Ubuntu package collection, any cleanly oriented sys admin would not install it in `/usr`, but rather would use local::lib to install it in the home directory. An Ubuntu Lucid Lynx admin would use the following to install the local::lib module under `/usr`:

INFO

[1] TEMPer USB Thermometer: *http://www.amazon.com/dp/B002VA813U*

[2] Listings for the article: *http://www.linux-magazine.com/Resources/Article-Code*

[3] "Cool Projects Edition" by Kyle Rankin, *Linux Journal* August 2010, pp. 32-34

[4] "Aberrant Behavior Detection in Time Series for Network Service Monitoring" by Jake D. Brutlag: *http://www.usenix.org/events/lisa00/brutlag.html*

[5] "A Signal Analysis of Network Traffic Anomalies" by Paul Barford, Jeffery Kline, David Plonka, and Amos Ron: *http://pages.cs.wisc.edu/~pb/paper_imw_02.pdf*

[6] "Traffic Anomaly Detection at Fine Time Scales with Bayes Nets" by Jeff Kline, Sangnam Nam, Paul Barford, David Plonka, and Amos Ron: *http://pages.cs.wisc.edu/~pb/icimp08_final.pdf*

[7] "libudev and Sysfs Tutorial" by Alan Ott: *http://www.signal11.us/oss/udev/*

```
sudo apt-get install liblocal-lib-perl
```

then run the CPAN shell with:

```
perl -Mlocal::lib -MCPAN -eshell
```

Typing

```
install Device::USB::PCSensor::HidTEMPer
```

within the CPAN shell starts the download and installs the module in the `perl5` subdirectory of the user's home directory. The script in Listing 1 looks for the module there, because of the `use local::lib` instruction.

Without any additional tricks, only `root` can read the sensor. However, unprivileged users can be allowed to extract the temperature values if the settings in Figure 6 are stored in a file named `99-tempsensor.rules` in the `/etc/udev/rules.d` directory.

Editing the rules file requires a restart of the `udev` subsystem with `sudo service restart udev`; after that, you're all set to start your new temperature measurement daemon. ■■■

LISTING 3: rrdtemp

```
001 #!/usr/bin/perl -w
002 ############################
003 # rrdlog - Graph Temperature
004 #          Data
005 # Mike Schilli, 2010
006 # (m@perlmeister.com)
007 ############################
008 use strict;
009 use local::lib;
010 use RRDTool::OO;
011 use
012   DateTime::Format::Strptime;
013
014 my $logfile = "temper.log";
015 my @data_points = ();
016 my $rrd_file    = "data.rrd";
017
018 my $date_fmt =
019   DateTime::Format::Strptime
020   ->new(
021  pattern =>
022    "%Y/%m/%d %H:%M:%S",
023  time_zone => "local",
024   );
025
026   # Read logged temp data
027 open FILE, "$logfile"
028   or die
029 "Cannot open $logfile ($!)";
030 while (<FILE>) {
031  if (/(.*) READ (.*)/) {
032   my ($datestr, $temp) =
033     ($1, $2);
034
035   my $dt =
036     $date_fmt
037     ->parse_datetime(
038    $datestr);
039   push @data_points,
040     [ $dt->epoch(), $temp ];
041  }
042 }
043 close FILE;
044
045   # Create RRD
046 my $rrd = RRDTool::OO->new(
047  file        => $rrd_file,
048  raise_error => 1,
049 );
050
051 my $rows = 60 * 24 * 30;
052
053 $rrd->create(
054  step        => 60 * 5,
055  data_source => {
056   name => "temp",
057   type => "GAUGE"
058  },
059  archive => {
060   rows    => $rows,
061   cpoints => 1,
062   cfunc   => 'AVERAGE',
063  },
064  start =>
065    $data_points[0]->[0] - 60,
066  hwpredict => {
067   rows             => $rows,
068   alpha            => 0.1,
069   beta             => 0.0035,
070   gamma            => 0.5,
071   seasonal_period => 24 *
072     60 /
073     5,
074   threshold     => 14,
075   window_length => 18,
076  },
077 );
078
079 for my $data_point (
080  @data_points)
081 {
082  $rrd->update(
083   time  => $data_point->[0],
084   value => $data_point->[1],
085  );
086 }
087
088   # Draw Graph
089 $rrd->graph(
090  image  => "bounds.png",
091  width  => 1600,
092  height => 800,
093  start =>
094    $data_points[0]->[0],
095  end =>
096    $data_points[-1]->[0],
097  draw => {
098   type  => "line",
099   color => '000000',
100   legend =>
101     "Temperature over Time",
102   thickness => 2,
103   cfunc     => 'AVERAGE',
104  },
105  draw => {
106   type   => "line",
107   color  => '00FF00',
108   cfunc  => 'HWPREDICT',
109   name   => 'predict',
110   legend => 'hwpredict',
111  },
112  draw => {
113   type  => "hidden",
114   cfunc => 'DEVPREDICT',
115   name  => 'dev',
116  },
117  draw => {
118   type  => "hidden",
119   name  => "failures",
120   cfunc => 'FAILURES',
121  },
122  tick => {
123   draw   => "failures",
124   color  => '#FF0000',
125   legend => "Failures",
126  },
127  draw => {
128   type   => "line",
129   color  => '0000FF',
130   legend => "Upper Bound",
131   cdef =>
132     "predict,dev,2,*,+",
133  },
134  draw => {
135   type   => "line",
136   color  => '0000FF',
137   legend => "Lower Bound",
138   cdef =>
139     "predict,dev,2,*,-",
140  },
141 );
```