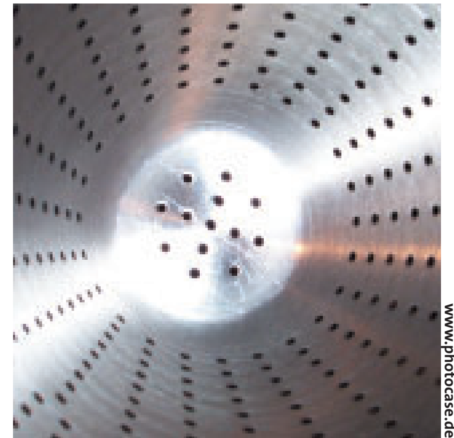
Handling data samples with RRDtool and Perl

Limiting Data

A steady flow of data samples does not automatically mean overflowing your hard disk. An overview is typically sufficient for older data, although you may need more detail on current events. BY MICHAEL SCHILLI



A round robin database like RRDtool will conveniently drop unimportant values, sticking forever with a fixed amount of storage.

Tobias Oetiker's RRDtool [2] has become a de-facto standard for storing network monitoring data. To do so, it uses a so-called round robin database (RRD) which front-ends such as Cacti access. Figure 1 should help you imagine what one of the round robin archives (RRA) of an RRD looks like.

As an example, it stores web server load values in a limited number of storage locations. The values start with 6.1 at 01:00 (top center), followed by a load of 2.0 at 01:01 through to 2.4 at 01:04, reading clockwise. The pointer indicates the last value updated.

At this point, the archive is full. For this reason, and as you can see in Figure 2, the value acquired at 01:00 is overwritten by a new value of 4.1 at 01:05. However, admins are not only interested in the values for the last five minutes. They need to see how the server load has developed over the past 30 days, or the last 12 months.

Fuzzy by Design

Again, there is no need to store enormous amounts of data to provide these statistics – a certain degree of fuzziness is acceptable for longer periods of time. The trick is to create additional RRAs which will store the average load (or the peak load, depending on your requirements) per hour, for the previous day, or per day in the current year.

After you have defined your round robin archives in a RRD database, you can use RRDtool to supply sampled values using either an explicit command, or the Perl interface provided. The underlying database engine ensures that data sent to the round robin database will update all relevant archives with their various degrees of granularity. When querying data for a given time frame later on, RRDtool will pick the archive with the best resolution for the window requested and return its data. It even draws neat graphs from it!

Using RRDtool in Perl Scripts

A round robin database includes a number of data sources (DS). The administrator supplies four parameters for each source when creating the database: a name, a data source type, the input step length, and the minimum and maximum input values.

The name (for example *load* or *mem_usage*) uniquely identifies the input data source in the RRD. Admins can use the data source type (DST) to specify whether input values should be recorded as is (*GAUGE*), or if they should be used to increment a *COUNTER*. RRDtool handles counter overflows gracefully by intercepting the overflow and correcting the result in the database.

If multiple values are acquired during a step, RRDtool calculates and stores the mean value. RRDtool stores *na* (not available) if there are no samples within the step, and ignores any values outside the minimum and maximum thresholds.

The following piece of Perl code creates a database which acquires entries from an input source called *load*. Every 60 seconds, the source will feed the CPU load value to the RRD:

```
use RRDs;

RRDs::create(
  "/tmp/load.rrd", "--step=60",
  "--start=" . time() - 10,
  "DS:load:GAUGE:90:0:10.0",
  "RRA:MAX:0.5:1:5",
  "RRA:MAX:0.5:5:10");
```

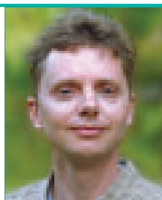
Unfortunately, RRDtool currently lacks an intuitive object-oriented interface. The slightly cryptic looking code will be explained in detail as we move on. In this example, RRDtool stores the database in a file called */tmp/load.rrd*. The interval at which data will be fed into the database is set to 60 seconds, using the *--step=60* option.

My Heart Goes Boom

The starting time for the database is set to 10 seconds in the past. This is common, and the default if you omit the *--start* parameter, as RRDtool rejects any input with a timestamp equal to or smaller than the starting time. The line starting with *DS:* above defines the single data source for the database: source name *load*, input type *GAUGE*, at a heartbeat of 90, and minimum and maximum thresholds of 0 and 10.0.

The heartbeat of 90 indicates that the admin is perfectly happy to receive input with a delay of up to 30 seconds on top of the 60 seconds set in the *--step* para-

THE AUTHOR



Michael Schilli works as a SW engineer for AOL/Netscape in Mountain View, California. He wrote "Perl Power" for Addison-Wesley and can be contacted at mschilli@perlmeister.com. His homepage is at http://perlmeister.com.

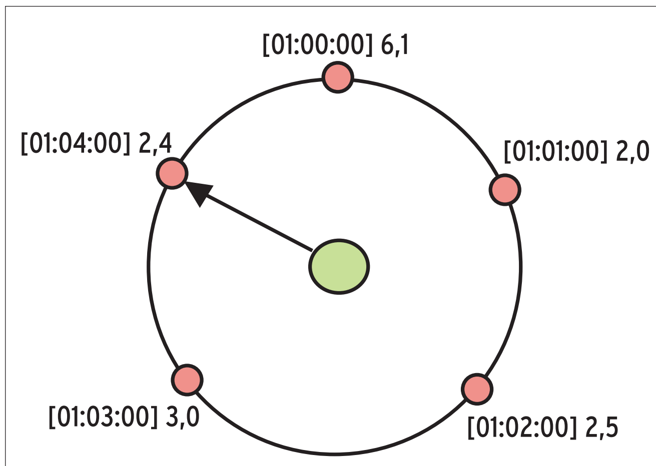


Figure 1: The round robin archive stores a fixed number of acquired values and overwrites older values to make room for newer ones.

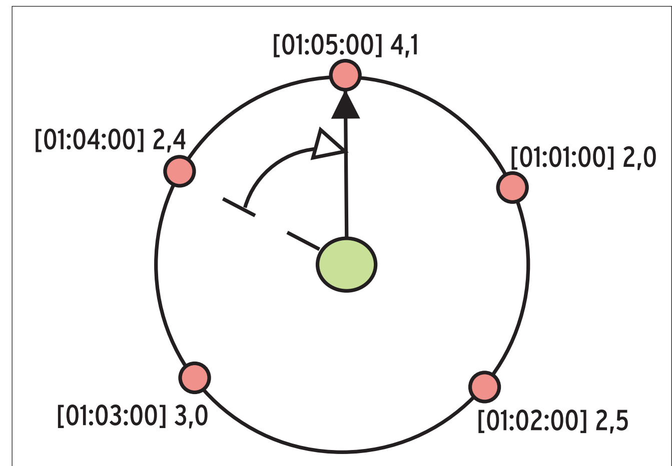


Figure 2: The old value, measured at 01:00 is replaced by a value for 01:05; the pointer moves one step farther.

meter. In this case, RRDtool will interpolate the data. If we were to set the heartbeat to 24 hours and maintain the step rate of 60, a single value per day would be all RRDtool needed to generate interpolated entries once a minute.

Primary Data Points

If the heartbeat is set to a lower value than the step rate, however, multiple values must be fed per step. In this case, RRDtool expects the data in the heartbeat frequency, and strictly assigns a grade of *na* if the step misses a beat. If everything goes as expected and multiple values are available per step window, RRDtool calculates the mean value before going on to save the so-called primary data point (PDP).

The last two lines in the code snippet above, each starting with "RRA:", create two different Round Robin Archives in which to store the data source's input. The number in the second-to-last column defines the number of PDPs that the archive will collate to form an archive point. The first archive shows a "1" and therefore simply uses a single value. Eventually, it will look exactly like the round robin archive shown in Figures 1 and 2, storing one value per minute.

The second archive, defined in the last line of the code snippet, shows a value of "5" and therefore collates five PDPs to form one archive point. The second column (right after "RRA") defines the consolidation function (CF). If the archive uses a 1:1 mapping between PDPs and archive points, this setting is irrelevant. If it collects more PDPs to form an archive point, however, it becomes critical: A setting of *AVERAGE* derives the mean value from the PDPs; the *MAX* function, on the other hand, takes the maximum value. Other options are *MIN* for the smallest and *LAST* for the latest acquired value.

The magic number *0.5* in the third column is referred to as the Xfiles Factor. It specifies what fraction of the PDPs can be undefined (*na*) for the archive to store the interpolated mean value as a valid entry. If this value is exceeded, the archive won't interpolate and will store *na*. The last column specifies the number of data slots the archive provides. If all of these are occupied, it will start to overwrite the oldest. Figure 3 shows you how RRDtool creates PDPs from the values provided by the data source, and how these values are then shifted to various round robin archives.

Revealing RRD's secrets

Listing 1 defines a test script that defines a RRD, inputs artificially generated values, and then queries the archive data. To allow it to provide reproducible results, the script uses a timestamp of *1080460200* rather than the system clock. Tip: RRD starts rounding if you do not use a number that is divisible by 60 (or 300 for the five minute archive). This will sort itself out in the end, but for the purpose of this demonstration, less unwieldy figures are preferable. Lines 19 through 24 create the RRD as described.

The *for* loop in line 28 runs from 0 through 40 and calls *RRDs::update()* to push the following timestamp/load value combinations to the RRD:

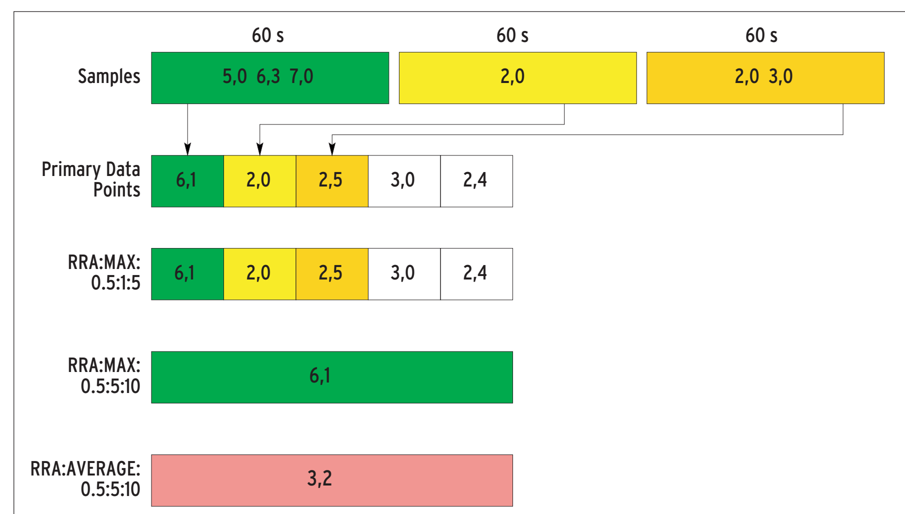


Figure 3: RRDtool uses the sample values from the data source to create primary data points (PDPs). RRDtool then uses PDPs to fill the round robin archive (RRA).

```
1080460200:2
1080460260:2.1
1080460320:2.2
...
```

There is no need to supply a timestamp for normal operations. If we provide the letter "N" instead of the number of seconds since the epoch, RRDtool will simply use the current system time. The values provided by the example are artificial values for the system load. The test script starts at *2*, and raises the value by *0.1* at each step.

Data Analysis

To query an archive, *RRDs::fetch()* expects the query interval, along with the CF used previously to define the database. It uses these parameters to determine the archive providing the best resolution for a given query. The module responds with an error message if you specify a CF for which there is no archive defined. In line 53, *RRDs::fetch()* retrieves the data points from the archive in *$data*, and returns them in a reference to an array, which in turn contains references that each point to an array of floating point values for a single archive point. The other return values for *RRDs::fetch()* are *$dbstart* (starting time for this RRD), *$step* (interval between data points in the archive), and *$names* (a reference to an array with the names of all data sources).

Incidentally, *$step* is not necessarily the data acquisition interval defined in the *--step* parameter for the database. For an archive that consolidates multiple PDPs to create an archive point, *$step* is calculated by multiplying the sample rate by the number of samples per archive point.

Line 39 in Listing 1 uses the *fetch()* function defined in line 47 to launch a query in the timeslot for the last five minutes. The results are as follows:

```
Last 5 minutes:
1080462300: N/A
1080462360: 5.6
1080462420: 5.7
1080462480: 5.8
1080462540: 5.9
1080462600: 6
```

In this case, RRDtool chooses the short-term archive with a 60 second sample rate. As the archive only holds five values, the oldest sample has been discarded and shows as *N/A* (for Not Applicable). If the user calls *RRDs::fetch()* to query values for a larger timeframe, the last 30 minutes, for example, as shown in line 43, values from the second archive with a sample rate of 300 seconds are returned instead:

```
Last 30 minutes:
1080460800: 3
1080461100: 3.5
1080461400: 4
1080461700: 4.5
1080462000: 5
1080462300: 5.5
1080462600: 6
```

The figures represent the maximum values for each interval, as the second archive was defined with the *MAX* CF. The *RRDs* module will not attempt to display values from a combination of archives. Instead, it selects a suitable archive and uses that archive's granularity to display a series of results with a constant sample rate.

The *rrdload* script in Listing 2 shows how a web server admin can put RRDtool to use. The following cronjob launches it every five minutes:

```
*/5 * * * * /home/mschilli/bin➘
/rrdload -u
```

When called with the *-u* option, it updates a round robin archive with the current system load value. Called with the *-g* option instead, it produces a

Listing 1: *rrdtest*

```
01 #!/usr/bin/perl
02 ###############################
03 # rrdtest - Feed test data
04 # Mike Schilli, 2004
05 # (m@perlmeister.com)
06 ###############################
07 use warnings;
08 use strict;
09
10 use RRDs;
11
12 my $DB   = "/tmp/mydemo.rrd";
13 my $start= 1080460200;
14 my $dst  = "MAX";
15 my $nof_iterations = 40;
16 my $end  = $start +
17         $nof_iterations * 60;
18
19 RRDs::create(
20   $DB, "--step=60",
21   "--start=" . ($start-10),
22   "DS:load:GAUGE:90:0:10.0",
23   "RRA:$dst:0.5:1:5",
24   "RRA:$dst:0.5:5:10",
25 ) or die "Cannot create " .
26      "rrd ($RRDs::error)";
27
28 for(0..$nof_iterations) {
29   my $time = $start + $_ *60;
30   my $value = 2 + $_ * 0.1;
31
32   RRDs::update(
33     $DB, "$time:$value") or
34       die "Can't update rrd".
35           " ($!)";
36 }
37
38 print "Last 5 minutes:\n";
39 fetch($end - 5*60,
40       $end, $dst);
41
42 print "Last 30 minutes:\n";
43 fetch($end - 30*60,
44       $end, $dst);
45
46 ###############################
47 sub fetch {
48 ###############################
49   my($start,$end, $dst) = @_;
50
51   my ($dbstart, $step,
52       $names, $data) =
53     RRDs::fetch($DB,
54       "--start=$start",
55       "--end=$end", $dst);
56
57   foreach my $row (@$data) {
58     print "$start: ";
59     $start += $step;
60     foreach my $val (@$row) {
61       $val = "N/A"
62          unless defined $val;
63       print "$val\n";
64     }
65   }
66 }
```

graphical representation of the relevant archive points. Figure 4 shows an example of a PNG formatted graph that the script stored in the Web server document path. This allows yours truly to monitor the system load on the shared system at perlmeister.com.

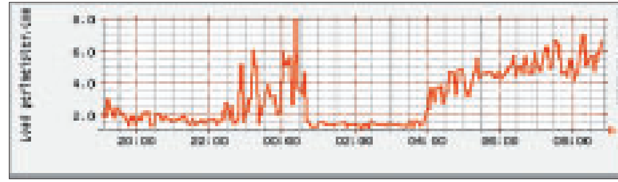


Figure 4: This graphic, which was created using RRDtool, shows the load on perlmeister.com's shared server over the course of one night.

Drawing Graphs

The code in line 20 ff. of Listing 2 creates three archives. The first archive can accommodate 288 data points; it has enough room to store values sampled every five minutes for a whole day (24·12). The second archive finds the peak value from twelve samples, that is for an hour's data (12·5 minutes = 60 minutes), and stores 168 of them. This allows users to query the hourly values for the past week (168 = 24·7). The third and last archive finds the daily peak values among 288 5-minute samples and stores 365 of them to create an annual statistic.

The script uses the *RRDs::graph()* function to create the PNG formatted graph. *--vertical-label* provides the label for the load axis. The following arguments:

```
"DEF:myload=$DB:load:MAX",
"LINE2:myload#FF0000"
```

tell the *RRDs* module to collect results from the file that *$DB* points to, and to assign them to the graph variable *myload*. This query searches for values generated by the data source *load*, in an archive that uses data collected via the *MAX* consolidation function for the period defined in lines 46 through 48 (*--start* to *--end*).

RRDtool has the unfortunate habit of filling the database with random values first, and then providing inaccurate information for the starting time. This is why the *rrd_start_time()* function defined in line 56 ff. keeps accessing the archive and skipping data until it comes up with something that makes sense. The function returns the date of the first sample, and the *graph* call starting in line 42 accepts it in line 47.

The *RRDs::fetch()* call in line 60 goes back exactly one day, if no additional parameters are set. If you want to view a graph for a longer period, you can specify *--start* to set a different starting point. Negative values set a time offset relative to the current time:

```
"--start", -365*24*3600
```

will display any available data, but with a minimum of granularity. In any case, the *graph* function draws a neat graph in red (*#FF0000*) and with a line width of exactly two pixels (*LINE2*). It saves the images in a PNG file named *load.png* in the web server's document path defined in *$SERVER*.

The *RRDs* Perl module, which uses RRDtool's shared library, is not available from CPAN, but it is provided with the RRD distribution. To install the module, load the latest tarball from [2], and enter *./configure; make* to run a build. The *RRDs.pm* distribution is located in the *perl-shared* subdirectory. Use *perl Makefile.PL; make install* to install. ■

Listing 2: *rrdload*

```
01 #!/usr/bin/perl
02 ###############################
03 # rrdload - Measure CPU Load
04 # Mike Schilli, 2004
05 # (m@perlmeister.com)
06 ###############################
07 use warnings;
08 use strict;
09
10 use RRDs;
11 use Getopt::Std;
12
13 getopts("ug", \my %opts);
14
15 my $DB     = "/tmp/load.rrd";
16 my $SERVER = "/www/htdocs";
17 my $UPTIME = "uptime";
18
19 if(! -f $DB) {
20   RRDs::create($DB,
21     "--step=300",
22     "DS:load:GAUGE:330:U:U",
23     "RRA:MAX:0.5:1:288",
24     "RRA:MAX:0.5:12:168",
25     "RRA:MAX:0.5:288:365",
26   ) or die "Create error: " .
27            "($RRDs::error)";
28 }
29
30 if(exists $opts{u}) {
31   my $uptime = `$UPTIME`;
32   my ($load) =
33     ($uptime =~ /(\d\.\d+)/);
34
35   RRDs::update($DB,
36     time() . ":$load") or
37       die "Update error: " .
38           "($RRDs::error)";
39 }
40
41 if(exists $opts{g}) {
42   RRDs::graph(
43    "$SERVER/load.png",
44    "--vertical-label=" .
45      "Load perlmeister.com",
46    "--start=" .
47      rrd_start_time(),
48    "--end=" . time(),
49    "DEF:myload=$DB:load:MAX",
50    "LINE2:myload#FF0000") or
51      die "graph failed " .
52          "($RRDs::error)";
53 }
54
55 ###############################
56 sub rrd_start_time {
57 ###############################
58   my ($start, $step, $names,
59       $data) =
60      RRDs::fetch($DB, "MAX");
61
62   foreach my $line (@$data) {
63     if(!defined $line->[0]) {
64       $start += $step;
65       next;
66     }
67     return $start;
68   }
69 }
```

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/44/Perl*

[2] RRDtool: *http://people.ee.ethz.ch/~oetiker/webtools/rrdtool/*