NAME

rrd−beginners − RRDtool Beginners' Guide

SYNOPSIS

Helping new RRDtool users to understand the basics of RRDtool

DESCRIPTION

This manual is an attempt to assist beginners in understanding the concepts of RRDtool. It sheds a light on differences between RRDtool and other databases. With help of an example, it explains the structure of RRDtool database. This is followed by an overview of the ''graph'' feature of RRDtool. At the end, it has sample scripts that illustrate the usage/wrapping of RRDtool within Shell or Perl scripts.

What makes RRDtool so special?

RRDtool is GNU licensed software developed by Tobias Oetiker, a system manager at the Swiss Federal Institute of Technology. Though it is a database, there are distinct differences between RRDtool databases and other databases as listed below:

- RRDtool stores data; that makes it a back-end tool. The RRDtool command set allows the creation of graphs; that makes it a front-end tool as well. Other databases just store data and can not create graphs.
- In case of linear databases, new data gets appended at the bottom of the database table. Thus its size keeps on increasing, whereas the size of an RRDtool database is determined at creation time. Imagine an RRDtool database as the perimeter of a circle. Data is added along the perimeter. When new data reaches the starting point, it overwrites existing data. This way, the size of an RRDtool database always remains constant. The name ''Round Robin'' stems from this behavior.
- Other databases store the values as supplied. RRDtool can be configured to calculate the rate of change from the previous to the current value and store this information instead.
- Other databases get updated when values are supplied. The RRDtool database is structured in such a way that it needs data at predefined time intervals. If it does not get a new value during the interval, it stores an UNKNOWN value for that interval. So, when using the RRDtool database, it is imperative to use scripts that run at regular intervals to ensure a constant data flow to update the RRDtool database.

RRDtool is designed to store time series of data. With every data update, an associated time stamp is stored. Time is always expressed in seconds passed since epoch (01−01−1970). RRDtool can be installed on Unix as well as Windows. It comes with a command set to carry out various operations on RRD databases. This command set can be accessed from the command line, as well as from Shell or Perl scripts. The scripts act as wrappers for accessing data stored in RRDtool databases.

Understanding by an example

The structure of an RRD database is different than other linear databases. Other databases define tables with columns, and many other parameters. These definitions sometimes are very complex, especially in large databases. RRDtool databases are primarily used for monitoring purposes and hence are very simple in structure. The parameters that need to be defined are variables that hold values and archives of those values. Being time sensitive, a couple of time related parameters are also defined. Because of its structure, the definition of an RRDtool database also includes a provision to specify specific actions to take in the absence of update values. Data Source (DS), heartbeat, Date Source Type (DST), Round Robin Archive (RRA), and Consolidation Function (CF) are some of the terminologies related to RRDtool databases.

The structure of a database and the terminology associated with it can be best explained with an example.

```
rrdtool create target.rrd \
        --start 1023654125 \
        --step 300 \
        DS:mem:GAUGE:600:0:671744 \
        RRA:AVERAGE:0.5:12:24 \
        RRA:AVERAGE:0.5:288:31
```

This example creates a database named *target.rrd*. Start time (1'023'654'125) is specified in total number of seconds since epoch (time in seconds since 01−01−1970). While updating the database, the update time is also specified. This update time MUST be large (later) then start time and MUST be in seconds since

epoch.

The step of 300 seconds indicates that database expects new values every 300 seconds. The wrapper script should be scheduled to run every **step** seconds so that it updates the database every **step** seconds.

DS (Data Source) is the actual variable which relates to the parameter on the device that is monitored. Its syntax is

```
DS:variable_name:DST:heartbeat:min:max
```

DS is a key word. `variable_name` is a name under which the parameter is saved in the database. There can be as many DSs in a database as needed. After every step interval, a new value of DS is supplied to update the database. This value is also called Primary Data Point **(PDP)**. In our example mentioned above, a new PDP is generated every 300 seconds.

Note, that if you do NOT supply new data points exactly every 300 seconds, this is not a problem, RRDtool will interpolate the data accordingly.

DST (Data Source Type) defines the type of the DS. It can be COUNTER, DERIVE, ABSOLUTE, GAUGE. A DS declared as COUNTER will save the rate of change of the value over a step period. This assumes that the value is always increasing (the difference between the current and the previous value is greater than 0). Traffic counters on a router are an ideal candidate for using COUNTER as DST. DERIVE is the same as COUNTER, but it allows negative values as well. If you want to see the rate of *change* in free disk space on your server, then you might want to use the DERIVE data type. ABSOLUTE also saves the rate of change, but it assumes that the previous value is set to 0. The difference between the current and the previous value is always equal to the current value. Thus it just stores the current value divided by the step interval (300 seconds in our example). GAUGE does not save the rate of change. It saves the actual value itself. There are no divisions or calculations. Memory consumption in a server is a typical example of gauge. The difference between the different types DSTs can be explained better with the following example:

```
Values       = 300, 600, 900, 1200
Step         = 300 seconds
COUNTER DS   =    1,  1,   1,    1
DERIVE DS    =    1,  1,   1,    1
ABSOLUTE DS  =    1,  2,   3,    4
GAUGE DS     = 300, 600, 900, 1200
```

The next parameter is **heartbeat**. In our example, heartbeat is 600 seconds. If the database does not get a new PDP within 300 seconds, it will wait for another 300 seconds (total 600 seconds). If it doesn't receive any PDP within 600 seconds, it will save an UNKNOWN value into the database. This UNKNOWN value is a special feature of RRDtool − it is much better than to assume a missing value was 0 (zero) or any other number which might also be a valid data value. For example, the traffic flow counter on a router keeps increasing. Lets say, a value is missed for an interval and 0 is stored instead of UNKNOWN. Now when the next value becomes available, it will calculate the difference between the current value and the previous value (0) which is not correct. So, inserting the value UNKNOWN makes much more sense here.

The next two parameters are the minimum and maximum value, respectively. If the variable to be stored has predictable maximum and minimum values, this should be specified here. Any update value falling out of this range will be stored as UNKNOWN.

The next line declares a round robin archive (RRA). The syntax for declaring an RRA is

```
RRA:CF:xff:step:rows
```

RRA is the keyword to declare RRAs. The consolidation function (CF) can be AVERAGE, MINIMUM, MAXIMUM, and LAST. The concept of the consolidated data point (CDP) comes into the picture here. A CDP is CFed (averaged, maximum/minimum value or last value) from *step* number of PDPs. This RRA will hold *rows* CDPs.

Lets have a look at the example above. For the first RRA, 12 (steps) PDPs (DS variables) are AVERAGEed (CF) to form one CDP. 24 (rows) of theses CDPs are archived. Each PDP occurs at 300 seconds. 12 PDPs represent 12 times 300 seconds which is 1 hour. It means 1 CDP (which is equal to 12 PDPs) represents data

worth 1 hour. 24 such CDPs represent 1 day (1 hour times 24 CDPs). This means, this RRA is an archive for one day. After 24 CDPs, CDP number 25 will replace the 1st CDP. The second RRA saves 31 CDPs; each CPD represents an AVERAGE value for a day (288 PDPs, each covering 300 seconds = 24 hours). Therefore this RRA is an archive for one month. A single database can have many RRAs. If there are multiple DSs, each individual RRA will save data for all the DSs in the database. For example, if a database has 3 DSs and daily, weekly, monthly, and yearly RRAs are declared, then each RRA will hold data from all 3 data sources.

Graphical Magic

Another important feature of RRDtool is its ability to create graphs. The ‘‘graph’’ command uses the ‘‘fetch’’ command internally to retrieve values from the database. With the retrieved values it draws graphs as defined by the parameters supplied on the command line. A single graph can show different DS (Data Sources) from a database. It is also possible to show the values from more than one database in a single graph. Often, it is necessary to perform some math on the values retrieved from the database before plotting them. For example, in SNMP replies, memory consumption values are usually specified in KBytes and traffic flow on interfaces is specified in Bytes. Graphs for these values will be more meaningful if values are represented in MBytes and mbps. The RRDtool graph command allows to define such conversions. Apart from mathematical calculations, it is also possible to perform logical operations such as greater than, less than, and if/then/else. If a database contains more than one RRA archive, then a question may arise − how does RRDtool decide which RRA archive to use for retrieving the values? RRDtool looks at several things when making its choice. First it makes sure that the RRA covers as much of the graphing time frame as possible. Second it looks at the resolution of the RRA compared to the resolution of the graph. It tries to find one which has the same or higher better resolution. With the ‘‘−r’’ option you can force RRDtool to assume a different resolution than the one calculated from the pixel width of the graph.

Values of different variables can be presented in 5 different shapes in a graph − AREA, LINE1, LINE2, LINE3, and STACK. AREA is represented by a solid colored area with values as the boundary of this area. LINE1/2/3 (increasing width) are just plain lines representing the values. STACK is also an area but it is ‘‘stack’’ed on top AREA or LINE1/2/3. Another important thing to note is that variables are plotted in the order they are defined in the graph command. Therefore care must be taken to define STACK only after defining AREA/LINE. It is also possible to put formatted comments within the graph. Detailed instructions can be found in the graph manual.

Wrapping RRDtool within Shell/Perl script

After understanding RRDtool it is now a time to actually use RRDtool in scripts. Tasks involved in network management are data collection, data storage, and data retrieval. In the following example, the previously created target.rrd database is used. Data collection and data storage is done using Shell scripts. Data retrieval and report generation is done using Perl scripts. These scripts are shown below:

Shell script (collects data, updates database)

```
#!/bin/sh
a=0
while [ "$a" == 0 ]; do
snmpwalk -c public 192.168.1.250 hrSWRunPerfMem > snmp_reply
    total_mem=`awk 'BEGIN {tot_mem=0}
                          { if ($NF == "KBytes")
                            {tot_mem=tot_mem+$(NF-1)}
                          }
                    END {print tot_mem}' snmp_reply`
    # I can use N as a replacement for the current time
    rrdtool update target.rrd N:$total_mem
    # sleep until the next 300 seconds are full
    perl -e 'sleep 300 - time % 300'
done # end of while loop
```

Perl script (retrieves data from database and generates graphs and statistics)

```
#!/usr/bin/perl -w
# This script fetches data from target.rrd, creates a graph of memory
# consumption on the target (Dual P3 Processor 1 GHz, 656 MB RAM)

# call the RRD perl module
use lib qw( /usr/local/rrdtool-1.0.41/lib/perl ../lib/perl );
use RRDs;
my $cur_time = time();                # set current time
my $end_time = $cur_time - 86400;     # set end time to 24 hours ago
my $start_time = $end_time - 2592000; # set start 30 days in the past

# fetch average values from the RRD database between start and end time
my ($start,$step,$ds_names,$data) =
    RRDs::fetch("target.rrd", "AVERAGE",
                "-r", "600", "-s", "$start_time", "-e", "$end_time");
# save fetched values in a 2-dimensional array
my $rows = 0;
my $columns = 0;
my $time_variable = $start;
foreach $line (@$data) {
  $vals[$rows][$columns] = $time_variable;
  $time_variable = $time_variable + $step;
  foreach $val (@$line) {
          $vals[$rows][++$columns] = $val;}
  $rows++;
  $columns = 0;
}
my $tot_time = 0;
my $count = 0;
# save the values from the 2-dimensional into a 1-dimensional array
for $i ( 0 .. $#vals ) {
    $tot_mem[$count] = $vals[$i][1];
    $count++;
}
my $tot_mem_sum = 0;
# calculate the total of all values
for $i ( 0 .. ($count-1) ) {
    $tot_mem_sum = $tot_mem_sum + $tot_mem[$i];
}
# calculate the average of the array
my $tot_mem_ave = $tot_mem_sum/($count);
# create the graph
RRDs::graph ("/images/mem_$count.png",
             "--title= Memory Usage",
             "--vertical-label=Memory Consumption (MB)",
             "--start=$start_time",
             "--end=$end_time",
             "--color=BACK#CCCCCC",
             "--color=CANVAS#CCFFFF",
             "--color=SHADEB#9999CC",
             "--height=125",
             "--upper-limit=656",
             "--lower-limit=0",
             "--rigid",
```

```
                "--base=1024",
                "DEF:tot_mem=target.rrd:mem:AVERAGE",
                "CDEF:tot_mem_cor=tot_mem,0,671744,LIMIT,UN,0,tot_mem,IF,1024,/",
                "CDEF:machine_mem=tot_mem,656,+,tot_mem,-",
                "COMMENT:Memory Consumption between $start_time",
                "COMMENT:    and $end_time                      ",
                "HRULE:656#000000:Maximum Available Memory - 656 MB",
                "AREA:machine_mem#CCFFFF:Memory Unused",
                "AREA:tot_mem_cor#6699CC:Total memory consumed in MB");
        my $err=RRDs::error;
        if ($err) {print "problem generating the graph: $err\n";}
        # print the output
        print "Average memory consumption is ";
        printf "%5.2f",$tot_mem_ave/1024;
        print " MB. Graphical representation can be found at /images/mem_$count.png.";
```

AUTHOR

Ketan Patel <k2pattu@yahoo.com>