NAME

rrdcreate − Set up a new Round Robin Database

SYNOPSIS

rrdtool create *filename* [**−−start|−b** *start time*] [**−−step|−s** *step*] [**−−template|−t** *template-file*] [**−−source|−r** *source-file*] [**−−no−overwrite|−O**] [**−−daemon|−d** *address*] [**DS:***ds-name*[**=***mapped-ds-name*[[*source-index*]]]**:***DST***:***dst arguments*] [**RRA:***CF***:***cf arguments*]

DESCRIPTION

The create function of RRDtool lets you set up new Round Robin Database (**RRD**) files. The file is created at its final, full size and filled with **UNKNOWN** data, unless one or more source **RRD** files have been specified and they hold suitable data to ''pre-fill'' the new **RRD** file.

filename

The name of the **RRD** you want to create. **RRD** files should end with the extension *.rrd*. However, **RRDtool** will accept any filename.

−−start|−b *start time* **(default: now − 10s)**

Specifies the time in seconds since 1970−01−01 UTC when the first value should be added to the **RRD**. **RRDtool** will not accept any data timed before or at the time specified.

See also ''AT-STYLE TIME SPECIFICATION'' in rrdfetch for other ways to specify time.

If one or more source files is used to pre-fill the new **RRD**, the **−−start** option may be omitted. In that case, the latest update time among all source files will be used as the last update time of the new **RRD** file, effectively setting the start time.

−−step|−s *step* **(default: 300 seconds)**

Specifies the base interval in seconds with which data will be fed into the **RRD**. A scaling factor may be present as a suffix to the integer; see ''STEP, HEARTBEAT, and Rows As Durations''.

−−no−overwrite|−O

Do not clobber an existing file of the same name.

−−daemon|−d *address*

Address of the rrdcached daemon. For a list of accepted formats, see the **−l** option in the rrdcached manual.

```
rrdtool create --daemon unix:/var/run/rrdcached.sock /var/lib/rrd/foo.rrd I<other o
```

[−−template|−t *template-file***]**

Specifies a template **RRD** file to take step, DS and RRA definitions from. This allows one to base the structure of a new file on some existing file. The data of the template file is NOT used for pre-filling, but it is possible to specify the same file as a source file (see below).

Additional DS and RRA definitions are permitted, and will be added to those taken from the template.

−−source|−r *source-file*

One or more source **RRD** files may be named on the command line. Data from these source files will be used to prefill the created **RRD** file. The output file and one source file may refer to the same file name. This will effectively replace the source file with the new **RRD** file. While there is the danger to loose the source file because it gets replaced, there is no danger that the source and the new file may be ''garbled'' together at any point in time, because the new file will always be created as a temporary file first and will only be moved to its final destination once it has been written in its entirety.

Prefilling is done by matching up DS names, RRAs and consolidation functions and choosing the best available data resolution when doing so. Prefilling may not be mathematically correct in all cases (e.g. if resolutions have to change due to changed stepping of the target RRD and old and new resolutions do not match up with old/new bin boundaries in RRAs).

In other words: A best effort is made to preserve data during prefilling. Also, pre-filling of RRAs may only be possible for certain kinds of DS types. Prefilling may also have strange effects on Holt-Winters forecasting RRAs. In other words: there is no guarantee for data-correctness.

When ''pre-filling'' a **RRD** file, the structure of the new file must be specified as usual using DS and RRA

specifications as outlined below. Data will be taken from source files based on DS names and types and in the order the source files are specified in. Data sources with the same name from different source files will be combined to form a new data source. Generally, for any point in time the new **RRD** file will cover after its creation, data from only one source file will have been used for pre-filling. However, data from multiple sources may be combined if it refers to different times or an earlier named source file holds unknown data for a time where a later one holds known data.

If this automatic data selection is not desired, the DS syntax allows one to specify a mapping of target and source data sources for prefilling. This syntax allows one to rename data sources and to restrict prefilling for a DS to only use data from a single source file.

Prefilling currently only works reliably for RRAs using one of the classic consolidation functions, that is one of: AVERAGE, MIN, MAX, LAST. It might also currently have problems with COMPUTE data sources.

Note that the act of prefilling during **create** is similar to a lot of the operations available via the **tune** command, but using **create** syntax.

DS:*ds-name***[=***mapped-ds-name***[[***source-index***]]]:***DST***:***dst arguments*

A single **RRD** can accept input from several data sources (**DS**), for example incoming and outgoing traffic on a specific communication line. With the **DS** configuration option you must define some basic properties of each data source you want to store in the **RRD**.

ds-name is the name you will use to reference this particular data source from an **RRD**. A *ds-name* must be 1 to 19 characters long in the characters [a−zA−Z0−9_].

DST defines the Data Source Type. The remaining arguments of a data source entry depend on the data source type. For GAUGE, COUNTER, DERIVE, DCOUNTER, DDERIVE and ABSOLUTE the format for a data source entry is:

DS:*ds-name***:**{*GAUGE* | *COUNTER* | *DERIVE* | *DCOUNTER* | *DDERIVE* | *ABSOLUTE*}**:***heartbeat***:***min***:***max*

For COMPUTE data sources, the format is:

DS:*ds-name***:***COMPUTE***:***rpn-expression*

In order to decide which data source type to use, review the definitions that follow. Also consult the section on ''HOW TO MEASURE'' for further insight.

GAUGE
: is for things like temperatures or number of people in a room or the value of a RedHat share.

COUNTER
: is for continuous incrementing counters like the ifInOctets counter in a router. The **COUNTER** data source assumes that the counter never decreases, except when a counter overflows. The update function takes the overflow into account. The counter is stored as a per-second rate. When the counter overflows, RRDtool checks if the overflow happened at the 32bit or 64bit border and acts accordingly by adding an appropriate value to the result.

DCOUNTER
: the same as **COUNTER**, but for quantities expressed as double-precision floating point number. Could be used to track quantities that increment by non-integer numbers, i.e. number of seconds that some routine has taken to run, total weight processed by some technology equipment etc. The only substantial difference is that **DCOUNTER** can either be upward counting or downward counting, but not both at the same time. The current direction is detected automatically on the second non-undefined counter update and any further change in the direction is considered a reset. The new direction is determined and locked in by the second update after reset and its difference to the value at reset.

DERIVE
: will store the derivative of the line going from the last to the current value of the data source. This can be useful for gauges, for example, to measure the rate of people entering or leaving a room. Internally, derive works exactly like COUNTER but without overflow checks. So if your counter does not reset at 32 or 64 bit you might want to use DERIVE and combine it with a MIN value of 0.

DDERIVE
the same as **DERIVE**, but for quantities expressed as double-precision floating point number.

NOTE on COUNTER vs DERIVE

by Don Baarda <don.baarda@baesystems.com>

If you cannot tolerate ever mistaking the occasional counter reset for a legitimate counter wrap, and would prefer ''Unknowns'' for all legitimate counter wraps and resets, always use DERIVE with min=0. Otherwise, using COUNTER with a suitable max will return correct values for all legitimate counter wraps, mark some counter resets as ''Unknown'', but can mistake some counter resets for a legitimate counter wrap.

For a 5 minute step and 32−bit counter, the probability of mistaking a counter reset for a legitimate wrap is arguably about 0.8% per 1Mbps of maximum bandwidth. Note that this equates to 80% for 100Mbps interfaces, so for high bandwidth interfaces and a 32bit counter, DERIVE with min=0 is probably preferable. If you are using a 64bit counter, just about any max setting will eliminate the possibility of mistaking a reset for a counter wrap.

ABSOLUTE
is for counters which get reset upon reading. This is used for fast counters which tend to overflow. So instead of reading them normally you reset them after every read to make sure you have a maximum time available before the next overflow. Another usage is for things you count like number of messages since the last update.

COMPUTE
is for storing the result of a formula applied to other data sources in the **RRD**. This data source is not supplied a value on update, but rather its Primary Data Points (PDPs) are computed from the PDPs of the data sources according to the rpn-expression that defines the formula. Consolidation functions are then applied normally to the PDPs of the COMPUTE data source (that is the rpn-expression is only applied to generate PDPs). In database software, such data sets are referred to as ''virtual'' or ''computed'' columns.

heartbeat defines the maximum number of seconds that may pass between two updates of this data source before the value of the data source is assumed to be **UNKNOWN**.

min and *max* define the expected range values for data supplied by a data source. If *min* and/or *max* are specified any value outside the defined range will be regarded as **UNKNOWN**. If you do not know or care about min and max, set them to U for unknown. Note that min and max always refer to the processed values of the DS. For a traffic−**COUNTER** type DS this would be the maximum and minimum data-rate expected from the device.

If information on minimal/maximal expected values is available, always set the min and/or max properties. This will help RRDtool in doing a simple sanity check on the data supplied when running update.

rpn-expression defines the formula used to compute the PDPs of a COMPUTE data source from other data sources in the same <RRD>. It is similar to defining a **CDEF** argument for the graph command. Please refer to that manual page for a list and description of RPN operations supported. For COMPUTE data sources, the following RPN operations are not supported: COUNT, PREV, TIME, and LTIME. In addition, in defining the RPN expression, the COMPUTE data source may only refer to the names of data source listed previously in the create command. This is similar to the restriction that **CDEF**s must refer only to **DEF**s and **CDEF**s previously defined in the same graph command.

When pre-filling the new **RRD** file using one or more source **RRD**s, the DS specification may hold an optional mapping after the DS name. This takes the form of an equal sign followed by a mapped-to DS name and an optional source index enclosed in square brackets.

For example, the DS

```
DS:a=b[2]:GAUGE:120:0:U
```

specifies that the DS named *a* should be pre-filled from the DS named *b* in the second listed source file

(source indices are 1−based).

RRA:*CF***:***cf arguments*

The purpose of an **RRD** is to store data in the round robin archives (**RRA**). An archive consists of a number of data values or statistics for each of the defined data-sources (**DS**) and is defined with an **RRA** line.

When data is entered into an **RRD**, it is first fit into time slots of the length defined with the **−s** option, thus becoming a *primary data point*.

The data is also processed with the consolidation function (*CF*) of the archive. There are several consolidation functions that consolidate primary data points via an aggregate function: **AVERAGE**, **MIN**, **MAX**, **LAST**.

AVERAGE
: the average of the data points is stored.

MIN
: the smallest of the data points is stored.

MAX
: the largest of the data points is stored.

LAST
: the last data points is used.

Note that data aggregation inevitably leads to loss of precision and information. The trick is to pick the aggregate function such that the *interesting* properties of your data is kept across the aggregation process.

The format of **RRA** line for these consolidation functions is:

RRA:{*AVERAGE* | *MIN* | *MAX* | *LAST*}**:***xff***:***steps***:***rows*

xff The xfiles factor defines what part of a consolidation interval may be made up from **UNKNOWN** data while the consolidated value is still regarded as known. It is given as the ratio of allowed **UNKNOWN** PDPs to the number of PDPs in the interval. Thus, it ranges from 0 to 1 (exclusive).

steps defines how many of these *primary data points* are used to build a *consolidated data point* which then goes into the archive. See also ''STEP, HEARTBEAT, and Rows As Durations''.

rows defines how many generations of data values are kept in an **RRA**. Obviously, this has to be greater than zero. See also ''STEP, HEARTBEAT, and Rows As Durations''.

Aberrant Behavior Detection with Holt-Winters Forecasting

In addition to the aggregate functions, there are a set of specialized functions that enable **RRDtool** to provide data smoothing (via the Holt-Winters forecasting algorithm), confidence bands, and the flagging aberrant behavior in the data source time series:

- **RRA:***HWPREDICT***:***rows***:***alpha***:***beta***:***seasonal period*[**:***rra-num*]
- **RRA:***MHWPREDICT***:***rows***:***alpha***:***beta***:***seasonal period*[**:***rra-num*]
- **RRA:***SEASONAL***:***seasonal period***:***gamma***:***rra-num*[**:smoothing−window=***fraction*]
- **RRA:***DEVSEASONAL***:***seasonal period***:***gamma***:***rra-num*[**:smoothing−window=***fraction*]
- **RRA:***DEVPREDICT***:***rows***:***rra-num*
- **RRA:***FAILURES***:***rows***:***threshold***:***window length***:***rra-num*

These **RRAs** differ from the true consolidation functions in several ways. First, each of the **RRA**s is updated once for every primary data point. Second, these **RRAs** are interdependent. To generate real-time confidence bounds, a matched set of SEASONAL, DEVSEASONAL, DEVPREDICT, and either HWPREDICT or MHWPREDICT must exist. Generating smoothed values of the primary data points requires a SEASONAL **RRA** and either an HWPREDICT or MHWPREDICT **RRA**. Aberrant behavior detection requires FAILURES, DEVSEASONAL, SEASONAL, and either HWPREDICT or MHWPREDICT.

The predicted, or smoothed, values are stored in the HWPREDICT or MHWPREDICT **RRA**. HWPREDICT and MHWPREDICT are actually two variations on the Holt-Winters method. They are interchangeable. Both

attempt to decompose data into three components: a baseline, a trend, and a seasonal coefficient. HWPREDICT adds its seasonal coefficient to the baseline to form a prediction, whereas MHWPREDICT multiplies its seasonal coefficient by the baseline to form a prediction. The difference is noticeable when the baseline changes significantly in the course of a season; HWPREDICT will predict the seasonality to stay constant as the baseline changes, but MHWPREDICT will predict the seasonality to grow or shrink in proportion to the baseline. The proper choice of method depends on the thing being modeled. For simplicity, the rest of this discussion will refer to HWPREDICT, but MHWPREDICT may be substituted in its place.

The predicted deviations are stored in DEVPREDICT (think a standard deviation which can be scaled to yield a confidence band). The FAILURES **RRA** stores binary indicators. A 1 marks the indexed observation as failure; that is, the number of confidence bounds violations in the preceding window of observations met or exceeded a specified threshold. An example of using these **RRAs** to graph confidence bounds and failures appears in rrdgraph.

The SEASONAL and DEVSEASONAL **RRAs** store the seasonal coefficients for the Holt-Winters forecasting algorithm and the seasonal deviations, respectively. There is one entry per observation time point in the seasonal cycle. For example, if primary data points are generated every five minutes and the seasonal cycle is 1 day, both SEASONAL and DEVSEASONAL will have 288 rows.

In order to simplify the creation for the novice user, in addition to supporting explicit creation of the HWPREDICT, SEASONAL, DEVPREDICT, DEVSEASONAL, and FAILURES **RRAs**, the **RRDtool** create command supports implicit creation of the other four when HWPREDICT is specified alone and the final argument *rra-num* is omitted.

rows specifies the length of the **RRA** prior to wrap around. Remember that there is a one-to-one correspondence between primary data points and entries in these RRAs. For the HWPREDICT CF, *rows* should be larger than the *seasonal period*. If the DEVPREDICT **RRA** is implicitly created, the default number of rows is the same as the HWPREDICT *rows* argument. If the FAILURES **RRA** is implicitly created, *rows* will be set to the *seasonal period* argument of the HWPREDICT **RRA**. Of course, the **RRDtool** *resize* command is available if these defaults are not sufficient and the creator wishes to avoid explicit creations of the other specialized function **RRAs**.

seasonal period specifies the number of primary data points in a seasonal cycle. If SEASONAL and DEVSEASONAL are implicitly created, this argument for those **RRAs** is set automatically to the value specified by HWPREDICT. If they are explicitly created, the creator should verify that all three *seasonal period* arguments agree.

alpha is the adaption parameter of the intercept (or baseline) coefficient in the Holt-Winters forecasting algorithm. See rrdtool for a description of this algorithm. *alpha* must lie between 0 and 1. A value closer to 1 means that more recent observations carry greater weight in predicting the baseline component of the forecast. A value closer to 0 means that past history carries greater weight in predicting the baseline component.

beta is the adaption parameter of the slope (or linear trend) coefficient in the Holt-Winters forecasting algorithm. *beta* must lie between 0 and 1 and plays the same role as *alpha* with respect to the predicted linear trend.

gamma is the adaption parameter of the seasonal coefficients in the Holt-Winters forecasting algorithm (HWPREDICT) or the adaption parameter in the exponential smoothing update of the seasonal deviations. It must lie between 0 and 1. If the SEASONAL and DEVSEASONAL **RRAs** are created implicitly, they will both have the same value for *gamma*: the value specified for the HWPREDICT *alpha* argument. Note that because there is one seasonal coefficient (or deviation) for each time point during the seasonal cycle, the adaptation rate is much slower than the baseline. Each seasonal coefficient is only updated (or adapts) when the observed value occurs at the offset in the seasonal cycle corresponding to that coefficient.

If SEASONAL and DEVSEASONAL **RRAs** are created explicitly, *gamma* need not be the same for both. Note that *gamma* can also be changed via the **RRDtool** *tune* command.

smoothing-window specifies the fraction of a season that should be averaged around each point. By default,

the value of *smoothing-window* is 0.05, which means each value in SEASONAL and DEVSEASONAL will be occasionally replaced by averaging it with its (*seasonal period**0.05) nearest neighbors. Setting *smoothing-window* to zero will disable the running-average smoother altogether.

rra-num provides the links between related **RRAs**. If HWPREDICT is specified alone and the other **RRAs** are created implicitly, then there is no need to worry about this argument. If **RRAs** are created explicitly, then carefully pay attention to this argument. For each **RRA** which includes this argument, there is a dependency between that **RRA** and another **RRA**. The *rra-num* argument is the 1−based index in the order of **RRA** creation (that is, the order they appear in the *create* command). The dependent **RRA** for each **RRA** requiring the *rra-num* argument is listed here:

- HWPREDICT *rra-num* is the index of the SEASONAL **RRA**.
- SEASONAL *rra-num* is the index of the HWPREDICT **RRA**.
- DEVPREDICT *rra-num* is the index of the DEVSEASONAL **RRA**.
- DEVSEASONAL *rra-num* is the index of the HWPREDICT **RRA**.
- FAILURES *rra-num* is the index of the DEVSEASONAL **RRA**.

threshold is the minimum number of violations (observed values outside the confidence bounds) within a window that constitutes a failure. If the FAILURES **RRA** is implicitly created, the default value is 7.

window length is the number of time points in the window. Specify an integer greater than or equal to the threshold and less than or equal to 28. The time interval this window represents depends on the interval between primary data points. If the FAILURES **RRA** is implicitly created, the default value is 9.

STEP, HEARTBEAT, and Rows As Durations

Traditionally RRDtool specified PDP intervals in seconds, and most other values as either seconds or PDP counts. This made the specification for databases rather opaque; for example

```
rrdtool create power.rrd \
  --start now-2h --step 1 \
  DS:watts:GAUGE:300:0:24000 \
  RRA:AVERAGE:0.5:1:864000 \
  RRA:AVERAGE:0.5:60:129600 \
  RRA:AVERAGE:0.5:3600:13392 \
  RRA:AVERAGE:0.5:86400:3660
```

creates a database of power values collected once per second, with a five minute (300 second) heartbeat, and four **RRA**s: ten days of one second, 90 days of one minute, 18 months of one hour, and ten years of one day averages.

Step, heartbeat, and PDP counts and rows may also be specified as durations, which are positive integers with a single-character suffix that specifies a scaling factor. See "rrd_scaled_duration" in librrd for scale factors of the supported suffixes: `s` (seconds), `m` (minutes), `h` (hours), `d` (days), `w` (weeks), `M` (months), and `y` (years).

Scaled step and heartbeat values (which are natively durations in seconds) are used directly, while consolidation function row arguments are divided by their step to produce the number of rows.

With this feature the same specification as above can be written as:

```
rrdtool create power.rrd \
  --start now-2h --step 1s \
  DS:watts:GAUGE:5m:0:24000 \
  RRA:AVERAGE:0.5:1s:10d \
  RRA:AVERAGE:0.5:1m:90d \
  RRA:AVERAGE:0.5:1h:18M \
  RRA:AVERAGE:0.5:1d:10y
```

The HEARTBEAT and the STEP

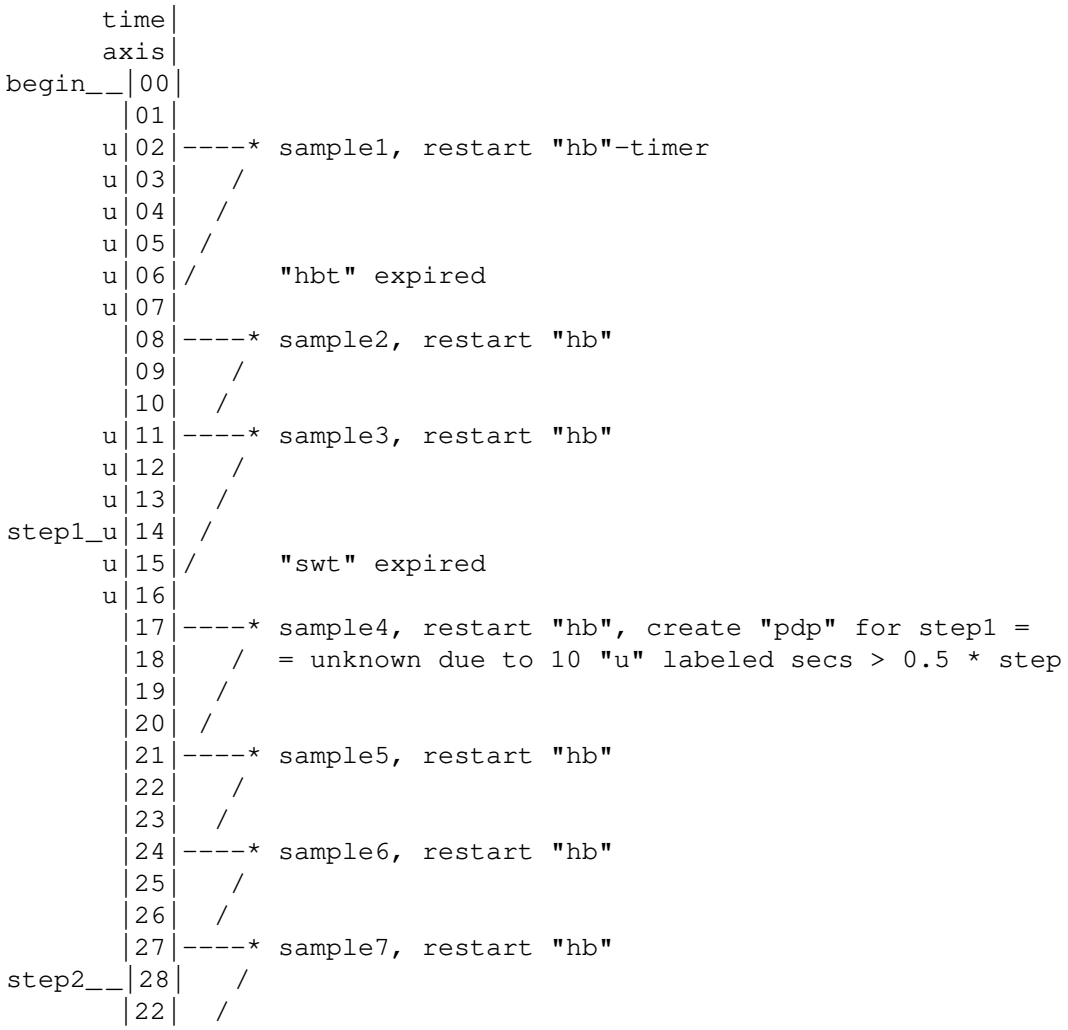
Here is an explanation by Don Baarda on the inner workings of RRDtool. It may help you to sort out why all this *UNKNOWN* data is popping up in your databases:

RRDtool gets fed samples/updates at arbitrary times. From these it builds Primary Data Points (PDPs) on every ''step'' interval. The PDPs are then accumulated into the RRAs.

The ''heartbeat'' defines the maximum acceptable interval between samples/updates. If the interval between samples is less than ''heartbeat'', then an average rate is calculated and applied for that interval. If the interval between samples is longer than ''heartbeat'', then that entire interval is considered ''unknown''. Note that there are other things that can make a sample interval ''unknown'', such as the rate exceeding limits, or a sample that was explicitly marked as unknown.

The known rates during a PDP's ''step'' interval are used to calculate an average rate for that PDP. If the total ''unknown'' time accounts for more than **half** the ''step'', the entire PDP is marked as ''unknown''. This means that a mixture of known and ''unknown'' sample times in a single PDP ''step'' may or may not add up to enough ''known'' time to warrant a known PDP.

The ''heartbeat'' can be short (unusual) or long (typical) relative to the ''step'' interval between PDPs. A short ''heartbeat'' means you require multiple samples per PDP, and if you don't get them mark the PDP unknown. A long heartbeat can span multiple ''steps'', which means it is acceptable to have multiple PDPs calculated from a single sample. An extreme example of this might be a ''step'' of 5 minutes and a ''heartbeat'' of one day, in which case a single sample every day will result in all the PDPs for that entire day period being set to the same average rate. -- *Don Baarda <don.baarda@baesystems.com>*



```
|23|----* sample8, restart "hb", create "pdp" for step1, create "cdp"
|24|   /
|25|  /
```

graphics by *vladimir.lavrov@desy.de*.

HOW TO MEASURE

Here are a few hints on how to measure:

Temperature

Usually you have some type of meter you can read to get the temperature. The temperature is not really connected with a time. The only connection is that the temperature reading happened at a certain time. You can use the **GAUGE** data source type for this. RRDtool will then record your reading together with the time.

Mail Messages

Assume you have a method to count the number of messages transported by your mail server in a certain amount of time, giving you data like '5 messages in the last 65 seconds'. If you look at the count of 5 like an **ABSOLUTE** data type you can simply update the RRD with the number 5 and the end time of your monitoring period. RRDtool will then record the number of messages per second. If at some later stage you want to know the number of messages transported in a day, you can get the average messages per second from RRDtool for the day in question and multiply this number with the number of seconds in a day. Because all math is run with Doubles, the precision should be acceptable.

It's always a Rate

RRDtool stores rates in amount/second for COUNTER, DERIVE, DCOUNTER, DDERIVE and ABSOLUTE data. When you plot the data, you will get on the y axis amount/second which you might be tempted to convert to an absolute amount by multiplying by the delta-time between the points. RRDtool plots continuous data, and as such is not appropriate for plotting absolute amounts as for example ''total bytes'' sent and received in a router. What you probably want is plot rates that you can scale to bytes/hour, for example, or plot absolute amounts with another tool that draws bar-plots, where the delta-time is clear on the plot for each point (such that when you read the graph you see for example GB on the y axis, days on the x axis and one bar for each day).

EXAMPLE

```
rrdtool create temperature.rrd --step 300 \
 DS:temp:GAUGE:600:-273:5000 \
 RRA:AVERAGE:0.5:1:1200 \
 RRA:MIN:0.5:12:2400 \
 RRA:MAX:0.5:12:2400 \
 RRA:AVERAGE:0.5:12:2400
```

This sets up an **RRD** called *temperature.rrd* which accepts one temperature value every 300 seconds. If no new data is supplied for more than 600 seconds, the temperature becomes **UNKNOWN**. The minimum acceptable value is −273 and the maximum is 5'000.

A few archive areas are also defined. The first stores the temperatures supplied for 100 hours (1'200 * 300 seconds = 100 hours). The second RRA stores the minimum temperature recorded over every hour (12 * 300 seconds = 1 hour), for 100 days (2'400 hours). The third and the fourth RRA's do the same for the maximum and average temperature, respectively.

EXAMPLE 2

```
rrdtool create monitor.rrd --step 300        \
  DS:ifOutOctets:COUNTER:1800:0:4294967295   \
  RRA:AVERAGE:0.5:1:2016                     \
  RRA:HWPREDICT:1440:0.1:0.0035:288
```

This example is a monitor of a router interface. The first **RRA** tracks the traffic flow in octets; the second **RRA** generates the specialized functions **RRAs** for aberrant behavior detection. Note that the *rra-num* argument of HWPREDICT is missing, so the other **RRAs** will implicitly be created with default parameter

values. In this example, the forecasting algorithm baseline adapts quickly; in fact the most recent one hour of observations (each at 5 minute intervals) accounts for 75% of the baseline prediction. The linear trend forecast adapts much more slowly. Observations made during the last day (at 288 observations per day) account for only 65% of the predicted linear trend. Note: these computations rely on an exponential smoothing formula described in the LISA 2000 paper.

The seasonal cycle is one day (288 data points at 300 second intervals), and the seasonal adaption parameter will be set to 0.1. The RRD file will store 5 days (1'440 data points) of forecasts and deviation predictions before wrap around. The file will store 1 day (a seasonal cycle) of 0–1 indicators in the FAILURES **RRA**.

The same RRD file and **RRAs** are created with the following command, which explicitly creates all specialized function **RRAs** using ''STEP, HEARTBEAT, and Rows As Durations''.

```
rrdtool create monitor.rrd --step 5m \
  DS:ifOutOctets:COUNTER:30m:0:4294967295 \
  RRA:AVERAGE:0.5:1:2016 \
  RRA:HWPREDICT:5d:0.1:0.0035:1d:3 \
  RRA:SEASONAL:1d:0.1:2 \
  RRA:DEVSEASONAL:1d:0.1:2 \
  RRA:DEVPREDICT:5d:5 \
  RRA:FAILURES:1d:7:9:5
```

Of course, explicit creation need not replicate implicit create, a number of arguments could be changed.

EXAMPLE 3

```
rrdtool create proxy.rrd --step 300 \
  DS:Requests:DERIVE:1800:0:U  \
  DS:Duration:DERIVE:1800:0:U  \
  DS:AvgReqDur:COMPUTE:Duration,Requests,0,EQ,1,Requests,IF,/ \
  RRA:AVERAGE:0.5:1:2016
```

This example is monitoring the average request duration during each 300 sec interval for requests processed by a web proxy during the interval. In this case, the proxy exposes two counters, the number of requests processed since boot and the total cumulative duration of all processed requests. Clearly these counters both have some rollover point, but using the DERIVE data source also handles the reset that occurs when the web proxy is stopped and restarted.

In the **RRD**, the first data source stores the requests per second rate during the interval. The second data source stores the total duration of all requests processed during the interval divided by 300. The COMPUTE data source divides each PDP of the AccumDuration by the corresponding PDP of TotalRequests and stores the average request duration. The remainder of the RPN expression handles the divide by zero case.

SECURITY

Note that new rrd files will have the permission 0644 regarless of your umask setting. If a file with the same name previously exists, its permission settings will be copied to the new file.

AUTHORS

Tobias Oetiker <tobi@oetiker.ch>, Peter Stamfest <peter@stamfest.at>