Tutorial

IMPORTANT: The following tutorial was taken from RRDTool's web site in its unchanged form. RRD4J related comments are placed in separate light-blue boxes like this one.

In April 2004. I asked Alex van den Bogaerdt (author of this tutorial) for his permission to mirror his tutorial here, with some comments on my own, but I got no response from him. His text was not changed in any way, so I hope he will approve this sooner or later. `0:-)`

04/05/2007: This tutorial has been updated for RRD4J version 2.0.2 by Watsh Rajneesh.

DESCRIPTION

RRDTool is written by Tobias Oetiker <oetiker@ee.ethz.ch> with contributions from many people all around the world. This document is written by Alex van den Bogaerdt <alex@ergens.op.het.net> to help you understand what RRDTool is and what it can do for you.

RRD4J is written by Sasa Markovic and Arne Vandamme as a pure Java alternative to Tobi's RRDTool set of command line utilities. RRD4J is a pure Java API (with a number of additional utilities) for RRD file handling and related graph creation.

The documentation provided with RRDTool can be too technical for some people. This tutorial is here to help you understand the basics of RRDTool. It should prepare you to read the documentation yourself. It also explains the general things about statistics with a focus on networking.

TUTORIAL by Alex van den Bogaerdt

Important

Please don't skip ahead in this document! The first part of this document explains the basics and may be boring. But if you don't understand the basics, the examples will not be as meaningful to you.

What is RRDTool ?

RRDTool refers to Round Robin Database tool. Round robin is a technique that works with a fixed amount of data, and a pointer to the current element. Think of a circle with some dots plotted on the edge, these dots are the places where data can be stored. Draw an arrow from the center of the circle to one of the dots, this is the pointer. When the current data is read or written, the pointer moves to the next element. As we are on a circle there is no beginning nor an end, you can go on and on. After a while, all the available places will be used and the process automatically reuses old locations. This way, the database will not grow in size and therefore requires no maintenance. RRDTool works with Round Robin Databases (RRDs). It stores and retrieves data from them.

What data can be put into an RRD ?

You name it, it will probably fit. You should be able to measure some value at several points in time and provide this information to RRDTool. If you can do this, RRDTool will be able to store it. The values need to be numerical but don't have to be, as opposed to MRTG, integers.

Many examples talk about SNMP which is an acronym for Simple Network Management Protocol. 'Simple' refers to the protocol -- it does not mean it is simple to manage or monitor a network. After working your way through this document, you should know enough to be able to understand what people are talking about. For now, just realize that SNMP is a way to ask devices for the values of counters they keep. It is the value from those counters that are kept in the RRD.

What can I do with this tool ?

RRDTool originated from MRTG (Multi Router Traffic Grapher). MRTG started as a tiny little script for graphing the use of a connection to the Internet. MRTG evolved into a tool for graphing other data sources including temperature, speed, voltage, number of printouts and the like. Most likely you will start to use the RRDTool to store and process data collected via SNMP. The data will most likely be bytes (or bits) transfered from and to a network or a computer. RRDTool lets you create a database, store data in it, retrieve that data and create graphs in GIF format for display on a web browser. Those GIF images are dependent on the data you collected and could be, for instance, an overview of the average network usage, or the peaks that occurred. It can also be used to display tidal waves, solar radiation, power consumption, number of visitors at an exhibition, noise levels near an airport, temperature on your favorite holiday location, temperature in the fridge and whatever you imagination can come up with. You need a sensor to measure the data and be able to feed the numbers to RRDTool.

What if I still have problems after reading this document ?

First of all: read it again! You may have missed something. If you are unable to compile the sources and you have a fairly common OS, it will probably not be the fault of RRDTool. There may be precompiled versions around on the Internet. If they come from trusted sources, get one of those. If on the other hand the program works but does not give you the expected results, it will be a problem with configuring it. Review your configuration and compare it with the examples that follow.

> Since RRD4J is written in 100% pure Java, there are no known compilation issues and OS related kind of problems. If you are able to compile and run Java code on your platform, you are ready to start using RRD4J right away. And RRD4J RRD files are fully portable: once created they can be easily copied and shared between platforms with different operating systems (the same is not true with RRDTool).

There is a mailing list and an archive of it. Read the list for a few weeks and search the archive. It is considered rude to just ask a question without searching the archives: your problem may already have been solved for somebody else! This is true for most, if not all, mailing lists and not only for this particular list! Look in the documentation that came with RRDTool for the location and usage of the list.

I suggest you take a moment to subscribe to the mailing list right now by sending an email to <rrd-users-request@list.ee.ethz.ch> with a subject of 'subscribe'. If you ever want to leave this list, you write an email to the same address but now with a subject of 'unsubscribe'.

> RRD4J has no mailing list for end-users. If you need help, post a question to our Forum site.

How will you help me ?

By giving you some detailed descriptions with detailed examples. It is assumed that following the instructions in the order presented will give you enough knowledge of RRDTool to experiment for yourself. If it doesn't work the first time, don't give up. Reread the stuff that you did understand, you may have missed something. By following the examples you get some hands-on experience and, even more important, some background information of how it works.

You will need to know something about hexadecimal numbers. If you don't then start with reading the bin_dec_hex manpage before you continue here.

Your first Round Robin Database

In my opinion the best way to learn something is to actually do it. Why not start right now? We will create a database, put some values in it and extract this data again. Your output should be the same as the output that is included in this document.

We will start with some easy stuff and compare a car with a router, or compare kilometers (miles if you wish) with bits and bytes. It's all the same: some number over some time.

Assume we have a device that transfers bytes to and from the Internet. This device keeps a counter that starts at zero when it is turned on, increasing with every byte that is transfered. This counter will have a maximum value, if that value is reached and an extra byte is counted, the counter starts all over at zero. This is the same as many counters in the world such as the mileage counter in a car. Most discussions about networking talk about bits per second so lets get used to that right away. Assume a byte is eight bits and start to think in bits not bytes. The counter, however, still counts bytes ! In the SNMP world most of the counters are 32 bits. That means they are counting from 0 to 4294967295. We will use these values in the examples. The device, when asked, returns the current value of the counter. We know the time that has passes since we last asked so we now know how many bytes have been transfered ***on average*** per second. This is not very hard to calculate. First in words, then in calculations:

- Take the current counter, subtract the previous value from it.
- Do the same with the current time and the previous time.
- Divide the outcome of (1) by the outcome of (2), the result is the amount of bytes per second.
- Multiply by eight to get the number of bits per second (bps).

```
bps = (counter_now - counter_before) / (time_now - time_before) * 8
```

For some people it may help to translate this to a automobile example: Do not try this example, and if you do, don't blame me for the results.

People who are not used to think in kilometers per hour can translate most into miles per hour by dividing km by 1.6 (close enough). I will use the following abbreviations:

```
M: meter
KM: kilometer (= 1000 meters).
H: hour
```

```
 S: second
 KM/H: kilometers per hour
 M/S: meters per second
```

You're driving a car. At 12:05 you read the counter in the dashboard and it tells you that the car has moved 12345 KM until that moment. At 12:10 you look again, it reads 12357 KM. This means you have traveled 12 KM in five minutes. A scientist would translate that into meters per second and this makes a nice comparison towards the problem of (bytes per five minutes) versus (bits per second).

We traveled 12 kilometers which is 12000 meters. We did that in five minutes which translates into 300 seconds. Our speed is 12000M / 300S equals 40 M/S.

We could also calculate the speed in KM/H: 12 times five minutes is an hour so we have to multiply 12 KM by 12 to get 144 KM/H. For our native English speaking friends: that's 90 MPH so don't try this example at home or where I live :)

Remember: these numbers are averages only. There is no way to figure out from the numbers, if you drove at a constant speed. There is an example later on in this tutorial that explains this.

I hope you understand that there is no difference in calculating M/S or bps; only the way we collect the data is different. Even the K from kilo is the same as in networking terms k also means 1000.

We will now create a database where we can keep all these interesting numbers. The method used to start the program may differ slightly from OS to OS but I assume you can figure it out if it works different on your OS. Make sure you do not overwrite any file on your system when executing the following command and type the whole line as one long line (I had to split it for readability) and skip all of the '\' characters.

```
rrdtool create test.rrd \
 --start 920804400 \
 DS:speed:COUNTER:600:U:U \
 RRA:AVERAGE:0.5:1:24 \
 RRA:AVERAGE:0.5:6:10
```

(So enter: `rrdtool create test.rrd --start 920804400 DS ...`)

In RRD4J:

```
RrdDef rrdDef = new RrdDef("./test.rrd");
rrdDef.setStartTime(920804400L);
rrdDef.addDatasource("speed", DsType.COUNTER, 600, Double.NaN, Double.NaN);
rrdDef.addArchive(ConsolFun.AVERAGE, 0.5, 1, 24);
rrdDef.addArchive(ConsolFun.AVERAGE, 0.5, 6, 10);
RrdDb rrdDb = new RrdDb(rrdDef);
rrdDb.close();
```

RRD4J RRD files are slightly smaller. The file created in the example above is 616 bytes long (compared with 1004 bytes for RRDTool on Linux).

What has been created ?

We created the round robin database called test (test.rrd) which starts at noon the day I started (7th of march, 1999) writing this document. It holds one data source (DS) named 'speed' that gets built from a counter. This counter is read every five minutes (default) In the same database two round robin archives (RRAs) are kept, one averages the data every time it is read (e.g. there's nothing to average) and keeps 24 samples (24 times 5 minutes is 2 hours). The other averages 6 values (half hour) and contains 10 of such averages (e.g. 5 hours) The remaining options will be discussed later on.

RRDTool works with special time stamps coming from the UNIX world. This time stamp is the number of seconds that passed since January 1st 1970 UTC. This time stamp is translated into local time and it will therefore look different for the different time zones.

Chances are that you are not in the same part of the world as I am. This means your time zone is different. In all examples where I talk about time, the hours may be wrong for you. This has little effect on the results of the examples, just correct the hours while reading. As an example: where I will see '12:05' the UK folks will see '11:05'.

We now have to fill our database with some numbers. We'll pretend to have read the following numbers:

```
 12:05 12345 KM
 12:10 12357 KM
 12:15 12363 KM
 12:20 12363 KM
 12:25 12363 KM
 12:30 12373 KM
```

```
12:35 12383 KM
12:40 12393 KM
12:45 12399 KM
12:50 12405 KM
12:55 12411 KM
13:00 12415 KM
13:05 12420 KM
13:10 12422 KM
13:15 12423 KM
```

We fill the database as follows:

```
rrdtool update test.rrd 920804700:12345 920805000:12357 920805300:12363
rrdtool update test.rrd 920805600:12363 920805900:12363 920806200:12373
rrdtool update test.rrd 920806500:12383 920806800:12393 920807100:12399
rrdtool update test.rrd 920807400:12405 920807700:12411 920808000:12415
rrdtool update test.rrd 920808300:12420 920808600:12422 920808900:12423
```

This reads: update our test database with the following numbers

```
time 920804700, value 12345
time 920805000, value 12357
```

etcetera.

In RRD4J:

```
RrdDb rrdDb = new RrdDb("./test.rrd");
Sample sample = rrdDb.createSample();
sample.setAndUpdate("920804700:12345");
sample.setAndUpdate("920805000:12357");
sample.setAndUpdate("920805300:12363");
sample.setAndUpdate("920805600:12363");
sample.setAndUpdate("920805900:12363");
sample.setAndUpdate("920806200:12373");
sample.setAndUpdate("920806500:12383");
sample.setAndUpdate("920806800:12393");
sample.setAndUpdate("920807100:12399");
sample.setAndUpdate("920807400:12405");
sample.setAndUpdate("920807700:12411");
sample.setAndUpdate("920808000:12415");
sample.setAndUpdate("920808300:12420");
sample.setAndUpdate("920808600:12422");
sample.setAndUpdate("920808900:12423");
rrdDb.close();
```

As you can see, it is possible to feed more than one value into the database in one command. I had to stop at three for readability but the real maximum is OS dependent.

We can now retrieve the data from our database using 'rrdtool fetch':

```
rrdtool fetch test.rrd AVERAGE --start 920804400 --end 920809200
```

It should return the following output:

```
speed
920804700: NaN
920805000: 0.04
920805300: 0.02
920805600: 0.00
920805900: 0.00
920806200: 0.03
920806500: 0.03
920806800: 0.03
920807100: 0.02
920807400: 0.02
920807700: 0.02
920808000: 0.01
920808300: 0.02
920808600: 0.01
```

```
920808900: 0.00
920809200: NaN
```

If it doesn't, something may be wrong. Perhaps your OS will print 'NaN' in a different form. It represents 'Not A Number'. If your OS writes 'U' or 'UNKN' or something similar that's okay. If something else is wrong, it will probably be due to an error you made (assuming that my tutorial is correct of course :-). In that case: delete the database and try again.

The following RRD4J code prints the same values to stdout:

```
RrdDb rrdDb = new RrdDb("./test.rrd");
FetchRequest fetchRequest = rrdDb.createFetchRequest(ConsolFun.AVERAGE, 920804400L, 920809200L);
FetchData fetchData = fetchRequest.fetchData();
fetchData.dump();
rrdDb.close();
```

What this output represents will become clear in the rest of the tutorial.

It is time to create some graphics. Try the following command:

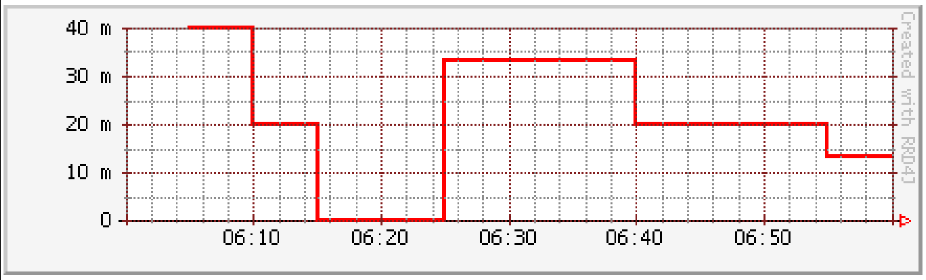
```
rrdtool graph speed.gif \
--start 920804400 --end 920808000 \
DEF:myspeed=test.rrd:speed:AVERAGE \
LINE2:myspeed#FF0000
```

This will create speed.gif which starts at 12:00 and ends at 13:00. There is a definition of variable myspeed, it is the data from RRA 'speed' out of database 'test.rrd'. The line drawn is 2 pixels high, and comes from variable myspeed. The color is red. You'll notice that the start of the graph is not at 12:00 but at 12:05 and this is because we have insufficient data to tell the average before that time. This will only happen when you miss some samples, this will not happen a lot, hopefully.

The following RRD4J code:

```
RrdGraphDef graphDef = new RrdGraphDef();
graphDef.setTimeSpan(920804400L, 920808000L);
graphDef.datasource("myspeed", "./test.rrd", "speed", ConsolFun.AVERAGE);
graphDef.line("myspeed", new Color(0xFF, 0, 0), null, 2);
graphDef.setFilename("./speed.gif");
RrdGraph graph = new RrdGraph(graphDef);
BufferedImage bi = new BufferedImage(100,100,BufferedImage.TYPE_INT_RGB);
graph.render(bi.getGraphics());
```

...creates the same graph:



GIF and JPEG image formats are supported in RRD4J, but only PNG format is recommended.

If this has worked: congratulations! If not, check what went wrong.

The colors are built up from red, green and blue. For each of the components, you specify how much to use in hexadecimal where 00 means not included and FF means fully included. The 'color' white is a mixture of red, green and blue: FFFFFF The 'color' black is all colors off: 000000

```
red #FF0000
green #00FF00
blue #0000FF
magenta #FF00FF (mixed red with blue)
gray #555555 (one third of all components)
```

RRD4J uses `java.awt.Color` class to represent colors:

```
red Color.RED
green Color.GREEN
blue Color.BLUE
magenta Color.MAGENTA
gray Color.GRAY
#000000 Color.BLACK
#99AABB new Color(0x99, 0xAA, 0xBB)
#112233 new Color(0x11, 0x22, 0x33)
```

The GIF you just created can be displayed using your favorite image viewer. Web browsers will display the GIF via the URL 'file://the/path/to/speed.gif'

Graphics with some math

When looking at the image, you notice that the horizontal axis is labeled 12:10, 12:20, 12:30, 12:40 and 12:50. The two remaining times (12:00 and 13:00) would not be displayed nicely so they are skipped. The vertical axis displays the range we entered. We provided kilometers and when divided by 300 seconds, we get very small numbers. To be exact, the first value was 12 (12357-12345) and divided by 300 this makes 0.04, which is displayed by RRDTool as '40 m' meaning '40/1000'. The 'm' has nothing to do with meters, kilometers or millimeters! RRDTool doesn't know about all this, it just works with numbers and not with meters...

What we did wrong was that we should have measured in meters, this would have been (12357000-12345000)/300 = 12000/300 = 40.

Let's correct that. We could recreate our database and store the correct data but there is a better way: do some calculations while creating the gif file !

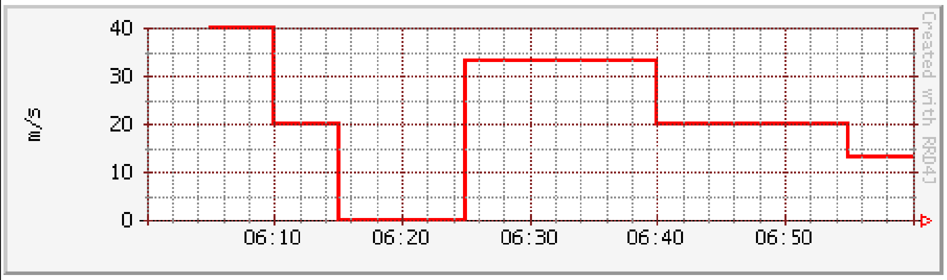
```
rrdtool graph speed2.gif \
 --start 920804400 --end 920808000 \
 --vertical-label m/s \
 DEF:myspeed=test.rrd:speed:AVERAGE \
 CDEF:realspeed=myspeed,1000,* \
 LINE2:realspeed#FF0000
```

After viewing this GIF, you notice the 'm' has disappeared. This it what the correct result would be. Also, a label has been added to the image. Apart from the things mentioned above, the GIF should be the same.

The following RRD4J code:

```
RrdGraphDef graphDef = new RrdGraphDef();
graphDef.setTimeSpan(920804400L, 920808000L);
graphDef.setVerticalLabel("m/s");
graphDef.datasource("myspeed", "./test.rrd", "speed", ConsolFun.AVERAGE);
graphDef.datasource("realspeed", "myspeed,1000,*");
graphDef.line("realspeed", new Color(0xFF, 0, 0), null, 2);
graphDef.setFilename("./speed2.gif");
graph = new RrdGraph(graphDef);
BufferedImage bi = new BufferedImage(100,100,BufferedImage.TYPE_INT_RGB);
graph.render(bi.getGraphics());
```

...creates the same GIF image:



The calculations are in the CDEF part and are in Reverse Polish Notation ('RPN'). What it says is: 'take the data source myspeed and the number 1000; multiply those'. Don't bother with RPN yet, it will be explained later on in more detail. Also, you may want to read my tutorial on CDEFs and Steve Rader's tutorial on RPN. But first finish this tutorial.

Hang on! If we can multiply values with 1000, it should also be possible to display kilometers per hour from the same data!

To change a value that is measured in meters per second:

```
Calculate meters per hour: value * 3600
Calculate kilometers per hour: value / 1000
Together this makes: value * (3600/1000) == value * 3.6
```

In our example database we made a mistake and we need to compensate for this by multiplying with 1000. Applying that correction:

```
value * 3.6 *1000 == value * 3600
```

Now let's create this GIF, and add some more magic ...

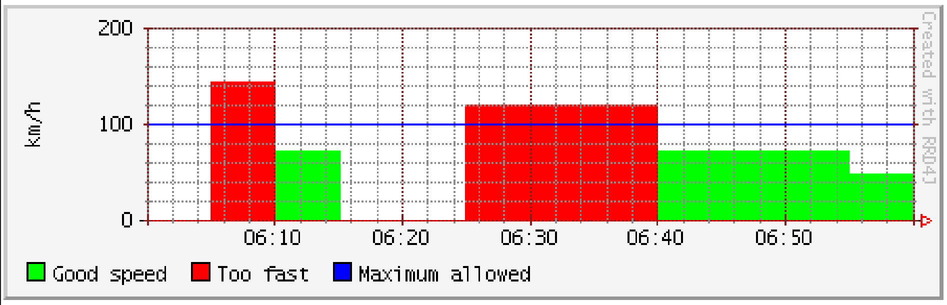
```
rrdtool graph speed3.gif \
 --start 920804400 --end 920808000 \
 --vertical-label km/h \
 DEF:myspeed=test.rrd:speed:AVERAGE \
 CDEF:kmh=myspeed,3600,* \
 CDEF:fast=kmh,100,GT,kmh,0,IF \
 CDEF:good=kmh,100,GT,0,kmh,IF \
 HRULE:100#0000FF:"Maximum allowed" \
 AREA:good#00FF00:"Good speed" \
 AREA:fast#FF0000:"Too fast"
```

This looks much better. Speed in KM/H and even an extra line with the maximum allowed speed (on the road I travel at). I also changed the colors used to display speed and changed it from a line into an area.

The following RRD4J code:

```
RrdGraphDef graphDef = new RrdGraphDef();
graphDef.setTimeSpan(920804400L, 920808000L);
graphDef.setVerticalLabel("km/h");
graphDef.datasource("myspeed", "./test.rrd", "speed", ConsolFun.AVERAGE);
graphDef.datasource("kmh", "myspeed,3600,*");
graphDef.datasource("fast", "kmh,100,GT,kmh,0,IF");
graphDef.datasource("good", "kmh,100,GT,0,kmh,IF");
graphDef.area("good", new Color(0, 0xFF, 0), "Good speed");
graphDef.area("fast", new Color(0xFF, 0, 0), "Too fast");
graphDef.hrule(100, new Color(0, 0, 0xFF), "Maximum allowed");
graphDef.setFilename("./speed3.gif");
graph = new RrdGraph(graphDef);
BufferedImage bi = new BufferedImage(100,100,BufferedImage.TYPE_INT_RGB);
graph.render(bi.getGraphics());
```

...creates the same GIF image:



The calculations are more complex now. For the 'good' speed they are:

```
Check if kmh is greater than 100 ( kmh,100 ) GT
If so, return 0, else kmh ((( kmh,100 ) GT ), 0, kmh) IF
```

For the other speed:

```
Check if kmh is greater than 100 ( kmh,100 ) GT
If so, return kmh, else return 0 ((( kmh,100) GT ), kmh, 0) IF
```

Graphics Magic

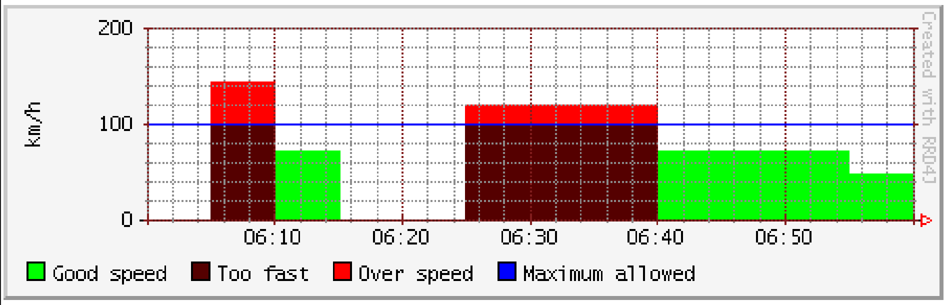
I like to believe there are virtually no limits to how RRDTool graph can manipulate data. I will not explain how it works, but look at the following GIF:

```
rrdtool graph speed4.gif \
 --start 920804400 --end 920808000 \
 --vertical-label km/h \
 DEF:myspeed=test.rrd:speed:AVERAGE \
 CDEF:kmh=myspeed,3600,* \
 CDEF:fast=kmh,100,GT,100,0,IF \
 CDEF:over=kmh,100,GT,kmh,100,-,0,IF \
 CDEF:good=kmh,100,GT,0,kmh,IF \
 HRULE:100#0000FF:"Maximum allowed" \
 AREA:good#00FF00:"Good speed" \
 AREA:fast#550000:"Too fast" \
 STACK:over#FF0000:"Over speed"
```

The following RRD4J code:

```
RrdGraphDef graphDef = new RrdGraphDef();
graphDef.setTimeSpan(920804400L, 920808000L);
graphDef.setVerticalLabel("km/h");
graphDef.datasource("myspeed", "./test.rrd", "speed", ConsolFun.AVERAGE);
graphDef.datasource("kmh", "myspeed,3600,*");
graphDef.datasource("fast", "kmh,100,GT,100,0,IF");
graphDef.datasource("over", "kmh,100,GT,kmh,100,-,0,IF");
graphDef.datasource("good", "kmh,100,GT,0,kmh,IF");
graphDef.area("good", new Color(0, 0xFF, 0), "Good speed");
graphDef.area("fast", new Color(0x55, 0, 0), "Too fast");
graphDef.stack("over", new Color(0xFF, 0, 0), "Over speed");
graphDef.hrule(100, new Color(0, 0, 0xFF), "Maximum allowed");
graphDef.setFilename("./speed4.gif");
graph = new RrdGraph(graphDef);
BufferedImage bi = new BufferedImage(100,100,BufferedImage.TYPE_INT_RGB);
graph.render(bi.getGraphics());
```

...creates the same GIF image:



Let's create a quick and dirty HTML page to view three GIFs:

```
<HTML>
<HEAD><TITLE>Speed</TITLE></HEAD>
<BODY>
<IMG src="speed2.gif" alt="Speed in meters per second"><BR>
<IMG src="speed3.gif" alt="Speed in kilometers per hour"><BR>
<IMG src="speed4.gif" alt="Traveled too fast?">
</BODY>
</HTML>
```

Name the file 'speed.html' or similar, and view it.

Now, all you have to do is measure the values regularly and update the database. When you want to view the data, recreate the GIFs and make sure to refresh them in your browser. (Note: just clicking reload may not be enough; Netscape in particular has a problem doing so and you'll need to click reload while pressing the shift key).

Updates in Reality

We've already used the 'update' command: it took one or more parameters in the form of '<time>:<value>'. You'll be glad to know that you can get the current time by filling in a 'N' as the time. If you wish, you can also use the 'time' function in Perl. The shortest example in this doc :)

```
perl -e 'print time, "\n" '
```

How you can run a program on regular intervals is OS specific. But here's an example in pseudo code:

```
Get the value, put it in variable "$speed"
rrdtool update speed.rrd N:$speed
```

(Do not try this with our test database, it is used in further examples)

This is all. Run this script every five minutes. When you need to know what the graphics look like, run the examples above. You could put them in a script. After running that script, view index.html

Some words on SNMP

I can imagine very few people will be able to get real data from their car every five minutes, all other people will have to settle for some other kind of counter. You could measure the number of pages printed by a printer, the coffee made by the coffee machine, a device that counts the electricity used, whatever. Any incrementing counter can be monitored and graphed using the stuff you learned until now. Later on we will also be able to monitor other types of values like temperature. Most people will use the counter that keeps track of octets (bytes) transfered by a network device so we have to do just that. We will start with a description of how to collect data. Some people will make a remark that there are tools who can do this data collection for you. They are right! However, I feel it is important that you understand they are not necessary. When you have to determine why things went wrong you need to know how they work.

One tool used in the example has been talked about very briefly in the beginning of this document, it is called SNMP. It is a way of talking to equipment. The tool I use below is called 'snmpget' and this is how it works:

```
snmpget device password OID
```

For device you substitute the name, or the IP address, of your device. For password you use the 'community read string' as it is called in the SNMP world. For some devices the default of 'public' might work, however this can be disabled, altered or protected for privacy and security reasons. Read the documentation that comes with your device or program.

Then there is this third parameter, called OID, which means 'object identifier'.

When you start to learn about SNMP it looks very confusing. It isn't all that difficult when you look at the Management Information Base ('MIB'). It is an upside-down tree that describes data, with a single node as the root and from there a number of branches. These branches end up in another node, they branch out, etc. All the branches have a name and they form the path that we follow all the way down. The branches that we follow are named: iso, org, dod, internet, mgmt and mib-2. These names can also be written down as numbers and are 1 3 6 1 2 1.

```
iso.org.dod.internet.mgmt.mib-2 (1.3.6.1.2.1)
```

There is a lot of confusion about the leading dot that some programs use. There is *no* leading dot in an OID. However, some programs can use above part of OIDs as a default. To indicate the difference between abbreviated OIDs and full OIDs they need a leading dot when you specify the complete OID. Often those programs will leave out the default portion when returning the data to you. To make things worse, they have several default prefixes ...

Right, lets continue to the start of our OID: we had 1.3.6.1.2.1 From there, we are especially interested in the branch 'interfaces' which has number 2 (e.g. 1.3.6.1.2.1.2 or 1.3.6.1.2.1.interfaces).

First, we have to get some SNMP program. First look if there is a pre-compiled package available for your OS. This is the preferred way. If not, you will have to get yourself the sources and compile those. The Internet is full of sources, programs etc. Find information using a search engine or whatever you prefer. As a suggestion: look for CMU-SNMP. It is commonly used.

Assume you got the program. First try to collect some data that is available on most systems. Remember: there is a short name for the part of the tree that interests us most in the world we live in!

I will use the short version as I think this document is large enough as it is. If that doesn't work for you, prefix with .1.3.6.1.2.1 and try again. Also, Read The Fine Manual. Skip the parts you cannot understand yet, you should be able to find out how to start the program and use it.

```
snmpget myrouter public system.sysDescr.0
```

The device should answer with a description of itself, perhaps empty. Until you got a valid answer from a device, perhaps using a different 'password', or a different device, there is no point in continuing.

```
snmpget myrouter public interfaces.ifNumber.0
```

Hopefully you get a number as a result, the number of interfaces. If so, you can carry on and try a different program called 'snmpwalk'.

```
snmpwalk myrouter public interfaces.ifTable.ifEntry.ifDescr
```

If it returns with a list of interfaces, you're almost there. Here's an example:

```
[user@host /home/alex]$ snmpwalk cisco public 2.2.1.2
 interfaces.ifTable.ifEntry.ifDescr.1 = "BRI0: B-Channel 1"
 interfaces.ifTable.ifEntry.ifDescr.2 = "BRI0: B-Channel 2"
 interfaces.ifTable.ifEntry.ifDescr.3 = "BRI0" Hex: 42 52 49 30
 interfaces.ifTable.ifEntry.ifDescr.4 = "Ethernet0"
 interfaces.ifTable.ifEntry.ifDescr.5 = "Loopback0"
```

On this cisco equipment, I would like to monitor the 'Ethernet0' interface and see that it is number four. I try:

```
[user@host /home/alex]$ snmpget cisco public 2.2.1.10.4 2.2.1.16.4
 interfaces.ifTable.ifEntry.ifInOctets.4 = 2290729126
 interfaces.ifTable.ifEntry.ifOutOctets.4 = 1256486519
```

So now I have two OIDs to monitor and they are (in full, this time):

```
1.3.6.1.2.1.2.2.1.10
```

and

```
1.3.6.1.2.1.2.2.1.16
```

both with an interface number of 4.

Don't get fooled, this wasn't my first try. It took some time for me too to understand what all these numbers mean, it does help a lot when they get translated into descriptive text... At least, when people are talking about MIBs and OIDs you know what it's all about. Do not forget the interface number (0 if it is not interface dependent) and try snmpwalk if you don't get an answer from snmpget.

If you understand above part, and get numbers from your device, continue on with this tutorial. If not, then go back and re-read this part.

A Real World Example

Let the fun begin. First, create a new database. It contains data from two counters, called input and output. The data is put into archives that average it. They take 1, 6, 24 or 288 samples at a time. They also go into archives that keep the maximum numbers. This will be explained later on. The time in-between samples is 300 seconds, a good starting point, which is the same as five minutes.

```
1 sample "averaged" stays 1 period of 5 minutes
6 samples averaged become one average on 30 minutes
24 samples averaged become one average on 2 hours
288 samples averaged become one average on 1 day
```

Lets try to be compatible with MRTG: MRTG stores about the following amount of data:

```
600 5-minute samples: 2 days and 2 hours
600 30-minute samples: 12.5 days
600 2-hour samples: 50 days
732 1-day samples: 732 days
```

These ranges are appended so the total amount of data kept is approximately 797 days. RRDTool stores the data differently, it doesn't start the 'weekly' archive where the 'daily' archive stopped. For both archives the most recent data will be near 'now' and therefore we will need to keep more data than MRTG does!

We will need:

```
600 samples of 5 minutes (2 days and 2 hours)
700 samples of 30 minutes (2 days and 2 hours, plus 12.5 days)
775 samples of 2 hours (above + 50 days)
797 samples of 1 day (above + 732 days, rounded up to 797)

rrdtool create myrouter.rrd \
 DS:input:COUNTER:600:U:U \
 DS:output:COUNTER:600:U:U \
 RRA:AVERAGE:0.5:1:600 \
 RRA:AVERAGE:0.5:6:700 \
 RRA:AVERAGE:0.5:24:775 \
 RRA:AVERAGE:0.5:288:797 \
 RRA:MAX:0.5:1:600 \
 RRA:MAX:0.5:6:700 \
 RRA:MAX:0.5:24:775 \
 RRA:MAX:0.5:288:797
```

In RRD4J:

```
RrdDef rrdDef = new RrdDef("./myrouter.rrd");
rrdDef.addDatasource("input", DsType.COUNTER, 600, Double.NaN, Double.NaN);
rrdDef.addDatasource("output", DsType.COUNTER, 600, Double.NaN, Double.NaN);
rrdDef.addArchive(ConsolFun.AVERAGE, 0.5, 1, 600);
rrdDef.addArchive(ConsolFun.AVERAGE, 0.5, 6, 700);
rrdDef.addArchive(ConsolFun.AVERAGE, 0.5, 24, 775);
rrdDef.addArchive(ConsolFun.AVERAGE, 0.5, 288, 797);
rrdDef.addArchive(ConsolFun.MAX, 0.5, 1, 600);
rrdDef.addArchive(ConsolFun.MAX, 0.5, 6, 700);
rrdDef.addArchive(ConsolFun.MAX, 0.5, 24, 775);
rrdDef.addArchive(ConsolFun.MAX, 0.5, 288, 797);
RrdDb rrdDb = new RrdDb(rrdDef);
rrdDb.close();
```

...or even:

```
RrdDef rrdDef = new RrdDef("./myrouter.rrd");
rrdDef.addDatasource("DS:input:COUNTER:600:U:U");
rrdDef.addDatasource("DS:output:COUNTER:600:U:U");
rrdDef.addArchive("RRA:AVERAGE:0.5:1:600");
rrdDef.addArchive("RRA:AVERAGE:0.5:6:700");
rrdDef.addArchive("RRA:AVERAGE:0.5:24:775");
rrdDef.addArchive("RRA:AVERAGE:0.5:288:797");
rrdDef.addArchive("RRA:MAX:0.5:1:600");
rrdDef.addArchive("RRA:MAX:0.5:6:700");
rrdDef.addArchive("RRA:MAX:0.5:24:775");
rrdDef.addArchive("RRA:MAX:0.5:288:797");
RrdDb rrdDb = new RrdDb(rrdDef);
rrdDb.close();
```

End effect is the same, of course.

Next thing to do is collect data and store it. Here is an example. It is written partially in pseudo code so you will have to find out what to do exactly on your OS to make it work.

```
while not the end of the universe
do
 get result of
 snmpget router community 2.2.1.10.4
 into variable $in
 get result of
 snmpget router community 2.2.1.16.4
 into variable $out
 rrdtool update myrouter.rrd N:$in:$out
 wait for 5 minutes
done
```

Then, after collecting data for a day, try to create an image using:

```
rrdtool graph myrouter-day.gif --start -86400 \
 DEF:inoctets=myrouter.rrd:input:AVERAGE \
 DEF:outoctets=myrouter.rrd:output:AVERAGE \
 AREA:inoctets#00FF00:"In traffic" \
 LINE1:outoctets#0000FF:"Out traffic" \
```

In RRD4J:

```
RrdGraphDef graphDef = new RrdGraphDef();
long endTime = Util.getTime();
long startTime = endTime - (24*60*60L);
graphDef.setTimeSpan(startTime, endTime);
graphDef.datasource("inoctets", "./myrouter.rrd", "input", ConsolFun.AVERAGE);
graphDef.datasource("outoctets", "./myrouter.rrd", "output", ConsolFun.AVERAGE);
graphDef.area("inoctets", new Color(0, 0xFF, 0), "In traffic");
graphDef.line("outoctets", new Color(0, 0, 0xFF), "Out traffic", 1);
graphDef.setFilename("./myrouter-day.gif");
BufferedImage bi = new BufferedImage(100,100,BufferedImage.TYPE_INT_RGB);
RrdGraph graph = new RrdGraph(graphDef);
graph.render(bi.getGraphics());
```

This should produce a picture with one day worth of traffic. One day is 24 hours of 60 minutes of 60 seconds: 24*60*60=86400, we start at now minus 86400 seconds. We define (with DEFs) inoctets and outoctets as the average values from the database myrouter.rrd and draw an area for the 'in' traffic and a line for the 'out' traffic.

View the image and keep logging data for a few more days. If you like, you could try the examples from the test database and see if you can get various options and calculations working.

Suggestion: Display in bytes per second and in bits per second. Make the Ethernet graphics go red if they are over four megabits per second.

Consolidation Functions

A few paragraphs back I mentioned the possibility of keeping the maximum values instead of the average values. Let's go into this a bit more.

Recall all the stuff about the speed of the car. Suppose we drove at 144 KM/H during 5 minutes and then were stopped by the police for 25 minutes. At the end of the lecture we would take our laptop and create+view the image taken from the database. If we look at the second RRA we did create, we would have the average from 6 samples. The samples measured would be 144+0+0+0+0+0=144, divided by 30 minutes, corrected for the error by 1000, translated into KM/H, with a result of 24 KM/H. I would still get a ticket but not for speeding anymore :)

Obviously, in this case, we shouldn't look at the averages. In some cases they are handy. If you want to know how much KM you had traveled, the picture would be the right one to look at. On the other hand, for the speed that we traveled at, the maximum number seen is much more valuable. (later we will see more types)

It is the same for data. If you want to know the amount, look at the averages. If you want to know the rate, look at the maximum. Over time, they will grow apart more and more. In the last database we have created, there are two archives that keep data per day. The archive that keeps averages will show low numbers, the archive that shows maxima will have higher numbers. For my car this would translate in averages per day of 96/24=4 KM/H (as I travel about 94 kilometers on a day) during week days, and maximum of 120 KM/H on weekdays (my top speed that I reach every day).

Big difference. Do not look at the second graph to estimate the distances that I travel and do not look at the first graph to estimate my speed. This will work if the samples are close together, as they are in five minutes, but not if you average.

On some days, I go for a long ride. If I go across Europe and travel for over 12 hours, the first graph will rise to about 60 KM/H. The second one will show 180 KM/H. This means that I traveled a distance of 60 KM/H times 24 H = 1440 KM. I did this with a higher speed and a maximum around 180 KM/H. This doesn't mean that I traveled for 8 hours at a constant speed of 180 KM/H ! This is a real example: go with the flow through Germany (fast!) and stop a few times for gas and coffee. Drive slowly through Austria and the Netherlands. Be careful in the mountains and villages. If you would look at the graphs created from the five-minute averages you would get a totally different picture. You would see the same values on the average and maximum graphs (provided I measured every 300 seconds). You would be able to see when I stopped, when I was in top gear, when I drove over fast highways etc. The granularity of the data is much higher, so you can see more. However, this takes 12 samples per hour, or 288 values per day, so it would be too much to keep for a long period of time. Therefore we average it, eventually to one value per day. From this one value, we cannot see much detail.

Make sure you understand the last few paragraphs. There is no value in only a line and a few axis, you need to know what they mean and interpret the data in a good way. This is true for all data.

The biggest mistake you can make is to use the collected data for something that it is not suitable for. You would be better off if you would not have the graphics at all in that case.

Let's review what you now should know.

You now know how to create a database. You can put the numbers in it, get them out again by creating an image, do math on the data from the database and view the outcome instead of the raw data. You know about the difference between averages and maxima, and when to use which (or at least you have an idea).

RRDTool can do more than what we have learned up to now. Before you continue with the rest of this doc, I recommend that you reread from the start and try some modifications on the examples. Make sure you fully understand everything. It will be worth the effort and helps you not only with the rest of this doc but also in your day to day monitoring long after you read this introduction.

Data Source Types

All right, you feel like continuing. Welcome back and get ready for an increased speed in the examples and explanation.

You know that in order to view a counter over time, you have to take two numbers and divide the difference of them between the time lapsed. This makes sense for the examples I gave you but there are other possibilities. For instance, I'm able to retrieve the temperature from my router in three places namely the inlet, the so called hot-spot and the exhaust. These values are not counters. If I take the difference of the two samples and divide that by 300 seconds I would be asking for the temperature change per second. Hopefully this is zero! If not, the computer room is on fire :)

So, what can we do ? We can tell RRDTool to store the values we measure directly as they are (this is not entirely true but close enough). The graphs we make will look much better, they will show a rather constant value. I know when the router is busy (it works -> it uses more electricity -> it generates more heat -> the temperature rises). I know when the doors are left open (the room is cooled -> the warm air from the rest of the building flows into the computer room -> the inlet temperature rises) etc. The data type we use when creating the database before was counter, we now have a different data type and thus a different name for it. It is called GAUGE. There are more such data types:

```
- COUNTER we already know this one
- GAUGE we just learned this one
- DERIVE
- ABSOLUTE
```

The two new types are DERIVE and ABSOLUTE. Absolute can be used like counter with one difference: RRDTool assumes the counter is reset when it's read. That is: its delta is known without calculation by RRDTool whereas RRDTool needs to calculate it for the counter type. Example: our first example (12345, 12357, 12363, 12363) would read: unknown, 12, 6, 0. The rest of the calculations stay the same. The other one, derive, is like counter. Unlike counter, it can also decrease so it can have a negative delta. Again, the rest of the calculations stay the same.

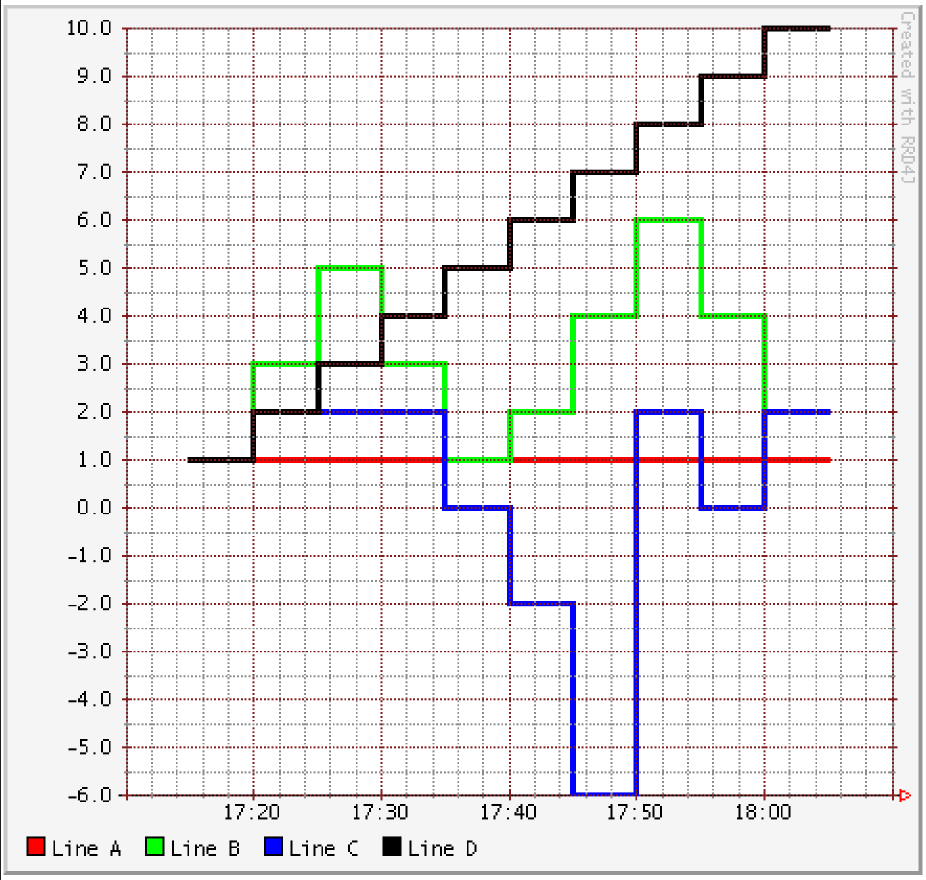
Let's try them all:

```
rrdtool create all.rrd --start 978300900 \
DS:a:COUNTER:600:U:U \
DS:b:GAUGE:600:U:U \
DS:c:DERIVE:600:U:U \
DS:d:ABSOLUTE:600:U:U \
RRA:AVERAGE:0.5:1:10
rrdtool update all.rrd \
978301200:300:1:600:300 \
978301500:600:3:1200:600 \
978301800:900:5:1800:900 \
978302100:1200:3:2400:1200 \
978302400:1500:1:2400:1500 \
978302700:1800:2:1800:1800 \
978303000:2100:4:0:2100 \
978303300:2400:6:600:2400 \
978303600:2700:4:600:2700 \
978303900:3000:2:1200:3000
rrdtool graph all1.gif -s 978300600 -e 978304200 -h 400 \
DEF:linea=all.rrd:a:AVERAGE LINE3:linea#FF0000:"Line A" \
DEF:lineb=all.rrd:b:AVERAGE LINE3:lineb#00FF00:"Line B" \
DEF:linec=all.rrd:c:AVERAGE LINE3:linec#0000FF:"Line C" \
DEF:lined=all.rrd:d:AVERAGE LINE3:lined#000000:"Line D"
```

The following RRD4J code:

```
RrdDef rrdDef = new RrdDef("./all.rrd");
rrdDef.setStartTime(978300900L);
rrdDef.addDatasource("a", DsType.COUNTER, 600, Double.NaN, Double.NaN);
rrdDef.addDatasource("b", DsType.GAUGE, 600, Double.NaN, Double.NaN);
rrdDef.addDatasource("c", DsType.DERIVE, 600, Double.NaN, Double.NaN);
rrdDef.addDatasource("d", DsType.ABSOLUTE, 600, Double.NaN, Double.NaN);
rrdDef.addArchive(ConsolFun.AVERAGE, 0.5, 1, 10);
RrdDb rrdDb = new RrdDb(rrdDef);
Sample sample = rrdDb.createSample();
sample.setAndUpdate("978301200:300:1:600:300");
sample.setAndUpdate("978301500:600:3:1200:600");
sample.setAndUpdate("978301800:900:5:1800:900");
sample.setAndUpdate("978302100:1200:3:2400:1200");
sample.setAndUpdate("978302400:1500:1:2400:1500");
sample.setAndUpdate("978302700:1800:2:1800:1800");
sample.setAndUpdate("978303000:2100:4:0:2100");
sample.setAndUpdate("978303300:2400:6:600:2400");
sample.setAndUpdate("978303600:2700:4:600:2700");
sample.setAndUpdate("978303900:3000:2:1200:3000");
rrdDb.close();
RrdGraphDef graphDef = new RrdGraphDef();
graphDef.setTimeSpan(978300600L, 978304200L);
graphDef.datasource("linea", "./all.rrd", "a", ConsolFun.AVERAGE);
graphDef.datasource("lineb", "./all.rrd", "b", ConsolFun.AVERAGE);
graphDef.datasource("linec", "./all.rrd", "c", ConsolFun.AVERAGE);
graphDef.datasource("lined", "./all.rrd", "d", ConsolFun.AVERAGE);
graphDef.line("linea", Color.RED, "Line A", 3);
graphDef.line("lineb", Color.GREEN, "Line B", 3);
graphDef.line("linec", Color.BLUE, "Line C", 3);
graphDef.line("lined", Color.BLACK, "Line D", 3);
graphDef.setFilename("./all1.gif");
graphDef.setWidth(400);
graphDef.setHeight(400);
RrdGraph graph = new RrdGraph(graphDef);
BufferedImage bim = new BufferedImage(400,400,BufferedImage.TYPE_INT_RGB);
graph.render(bim.getGraphics());
```

...produced the same graph:



RRDTool under the Microscope

Line A is a counter so it should continuously increment and RRDTool should calculate the differences. Also, RRDTool needs to divide the difference by the amount of time lapsed. This should end up as a straight line at 1 (the deltas are 300, the time is 300).

Line B is of type gauge. These are 'real' values so they should match what we put in: a sort of a wave.

Line C is derive. It should be a counter that can decrease. It does so between 2400 and 0, with 1800 in-between.

Line D is of type absolute. This is like counter but it works on values without calculating the difference. The numbers are the same and as you can see (hopefully) this has a different result.

This translates in the following values, starting at 23:10 and ending at 00:10 the next day (where U means unknown/unplotted):

```
- Line A: u u 1 1 1 1 1 1 1 1 1 u
- Line B: u 1 3 5 3 1 2 4 6 4 2 u
- Line C: u u 2 2 2 0 -2 -6 2 0 2 u
- Line D: u 1 2 3 4 5 6 7 8 9 10 u
```

If your GIF shows all this, you know you have typed the data correct, the RRDTool executable is working properly, your viewer doesn't fool you and you successfully entered the year 2000 :) You could try the same example four times, each time with only one of the lines.

Let's go over the data again:

Line A: 300,600,900 and so on. The counter delta is a constant 300 and so it the time delta. A number divided by itself is always 1 (except when dividing by zero which is undefined/illegal). Why is it that the first point is unknown ? We do know what we put into the database ? True ! But we didn't have a value to calculate the delta from so we don't know where we started. It would be wrong to assume we started at zero so we don't !

Line B: There is nothing to calculate. The numbers are as is.

Line C: Again, the start-out value is unknown. The same story is valid like for line A. In this case the deltas are not constant so the line is not. If we would put the same numbers in the database as we did for line A, we would have gotten the same line. Unlike type counter, this type can decrease and I hope to show you later on why there is a difference.

Line D: Here the device calculates the deltas. Therefore we DO know the first delta and it is plotted. We had the same input as with line A but the meaning of this input is different. Therefore the line is different. In this case the deltas increase each time with 300. The time delta stays at a constant 300 and therefore the division of the two gives increasing results.

Counter Wraps

There are a few more basics to show. Some important options are still to be covered and we haven't look at counter wraps yet. First the counter wrap: In our car we notice that our counter shows 999987. We travel 20 KM and the counter should go to 1000007. Unfortunately, there are only six digits on our counter so it really shows 000007. If we would plot that on a type DERIVE, it would mean that the counter was set back 999980 KM. It wasn't, and there has to be some protection for this. This protection is only available for type COUNTER which should be used for this kind of counter anyways. How does it work ? Type counter should never decrease and therefore RRDTool must assume it wrapped if it does decrease ! If the delta is negative, this can be compensated for by adding the maximum value of the counter + 1. For our car this would be:

```
Delta = 7 - 999987 = -999980 (instead of 1000007-999987=20)
Real delta = -999980 + 999999 + 1 = 20
```

At the time of writing this document, RRDTool knows of counters that are either 32 bits or 64 bits of size. These counters can handle the following different values:

```
- 32 bits: 0 .. 4294967295
- 64 bits: 0 .. 18446744073709551615
```

If these numbers look strange to you, you would like to view them in their hexadecimal form:

```
- 32 bits: 0 .. FFFFFFFF
- 64 bits: 0 .. FFFFFFFFFFFFFFFF
```

RRDTool handles both counters the same. If an overflow occurs and the delta would be negative, RRDTool first adds the maximum of a small counter + 1 to the delta. If the delta is still negative, it had to be the large counter that wrapped. Add the maximum possible value of the large counter + 1 and subtract the falsely added small value. There is a risk in this: suppose the large counter wrapped while adding a huge delta, it could happen in theory that adding the smaller value would make the delta positive. In this unlikely case the results would not be correct. The increase should be nearly as high as the maximum counter value for that to happen so chances are you would have several other problems as well and this particular problem would not even be worth thinking about. Even though I did include an example of it so you can judge that for yourself.

The next section gives you some numerical examples for counter-wraps. Try to do the calculations yourself or just believe me if your calculator can't handle the numbers :)

Correction numbers:

```
- 32 bits: (4294967295+1) = 4294967296
- 64 bits: (18446744073709551615+1)-correction1 = 18446744069414584320
Before: 4294967200
```

```
Increase: 100
Should become: 4294967300
But really is: 4
Delta: -4294967196
Correction1: -4294967196 +4294967296 = 100
Before: 18446744073709551000
Increase: 800
Should become: 18446744073709551800
But really is: 184
Delta: -18446744073709550816
Correction1: -18446744073709550816 +4294967296 = -18446744069414583520
Correction2: -18446744069414583520 +18446744069414584320 = 800
Before: 18446744073709551615 ( maximum value )
Increase: 18446744069414584320 ( absurd increase, minimum for
Should become: 36893488143124135935 this example to work )
But really is: 18446744069414584319
Delta: -4294967296
Correction1: -4294967296 + 4294967296 = 0
(not negative -> no correction2)
Before: 18446744073709551615 ( maximum value )
Increase: 18446744069414584319 ( one less increase )
Should become: 36893488143124135934
But really is: 18446744069414584318
Delta: -4294967297
Correction1: -4294967297 +4294967296 = -1
Correction2: -1 +18446744069414584320 = 18446744069414584319
```

As you can see from the last two examples, you need strange numbers for RRDTool to fail (provided it's bug free of course) so this should not happen. However, SNMP or whatever method you choose to collect the data might also report wrong numbers occasionally. We can't prevent all errors but there are some things we can do. The RRDTool 'create' command takes two special parameters for this. They define the minimum and maximum allowed value. Until now, we used 'U', meaning 'unknown'. If you provide values for one or both of them and if RRDTool receives values that are outside these limits, it will ignore those values. For a thermometer in degrees Celsius, the absolute minimum is just under -273. For my router, I can assume this minimum is much higher so I would say it is 10. The maximum temperature for my router I would state as 80. Any higher and the device would be out of order. For my car, I would never expect negative numbers and also I would not expect numbers to be higher than 230. Anything else, and there must have been an error. Remember: the opposite is not true, if the numbers pass this check it doesn't mean that they are correct. Always judge the graph with a healthy dose of paranoia if it looks weird.

Data Resampling

One important feature of RRDTool has not been explained yet: It is virtually impossible to collect the data and feed it into RRDTool on exact intervals. RRDTool therefore interpolates the data so it is on exact intervals. If you do not know what this means or how it works, then here's the help you seek:

Suppose a counter increases with exactly one for every second. You want to measure it in 300 seconds intervals. You should retrieve values that are exactly 300 apart. However, due to various circumstances you are a few seconds late and the interval is 303. The delta will also be 303 in that case. Obviously RRDTool should not put 303 in the database and make you believe that the counter increased 303 in 300 seconds. This is where RRDTool interpolates: it alters the 303 value as if it would have been stored earlier and it will be 300 in 300 seconds. Next time you are at exactly the right time. This means that the current interval is 297 seconds and also the counter increased with 297. Again RRDTool alters the value and stores 300 as it should be.

```
in the RRD in reality
time+000: 0 delta="U" time+000: 0 delta="U"
time+300: 300 delta=300 time+300: 300 delta=300
time+600: 600 delta=300 time+603: 603 delta=303
time+900: 900 delta=300 time+900: 900 delta=297
```

Let's create two identical databases. I've chosen the time range 920805000 to 920805900 as this goes very well with the example numbers.

```
rrdtool create seconds1.rrd \
--start 920804700 \
DS:seconds:COUNTER:600:U:U \
RRA:AVERAGE:0.5:1:24
for Unix: cp seconds1.rrd seconds2.rrd
for Dos: copy seconds1.rrd seconds2.rrd
for vms: how would I know :)
```

```
rrdtool update seconds1.rrd \
920805000:000 920805300:300 920805600:600 920805900:900
rrdtool update seconds2.rrd \
920805000:000 920805300:300 920805603:603 920805900:900
rrdtool graph seconds1.gif \
--start 920804700 --end 920806200 \
--height 200 \
--upper-limit 1.05 --lower-limit 0.95 --rigid \
DEF:seconds=seconds1.rrd:seconds:AVERAGE \
CDEF:unknown=seconds,UN \
LINE2:seconds#0000FF \
AREA:unknown#FF0000
rrdtool graph seconds2.gif \
--start 920804700 --end 920806200 \
--height 200 \
--upper-limit 1.05 --lower-limit 0.95 --rigid \
DEF:seconds=seconds2.rrd:seconds:AVERAGE \
CDEF:unknown=seconds,UN \
LINE2:seconds#0000FF \
AREA:unknown#FF0000
```

Both graphs should show the same.

It's up to you to check that, in this issue, RRD4J follows the same logic as RRDTool. If you followed the tutorial this far, you should be able to prove this easily.

WRAPUP

It's time to wrap up this document. You now know all the basics to be able to work with RRDTool and to read the documentation available. There is plenty more to discover about RRDTool and you will find more and more uses for the package. You could create easy graphics using just the examples provided and using only RRDTool. You could also use the front ends that are available.

MAILINGLIST

Remember to subscribe to the mailing-list. Even if you are not answering the mails that come by, it helps both you and the rest. A lot of the stuff that I know about MRTG (and therefore about RRDTool) I've learned while just reading the list without posting to it. I did not need to ask the basic questions as they are answered in the FAQ (read it!) and in various mails by other users. With thousands of users all over the world, there will always be people who ask questions that you can answer because you read this and other documentation and they didn't.

As I mentioned before, RRD4J has no mailing list for end users, but all RRD4J-related questions will be answered (probably) on our Forum site .

SEE ALSO

The RRDTool manpages.

AUTHOR

I hope you enjoyed the examples and their descriptions. If you do, help other people by pointing them to this document when they are asking basic questions. They will not only get their answer but at the same time learn a whole lot more.

`Alex van den Bogaerdt <alex@ergens.op.het.net>`

Back to the top