Dstat: plugin-based real-time monitoring

Table of Contents

1. Introduction

Many tools exist to monitor hardware resources and software behaviour, but few tools exist that allow you to easily monitor any conceivable counter.

Dstat was designed with the idea that it should be simple to plug in a piece of code that extracts one or more counters, and make it visible in a way that visually pleases the eye and helps you extract information in real-time.

By being able to select those counters that you want (and likely those counters that matter to you in the job you're doing) you make it easier to correlate raw numbers and see a pattern that may otherwise not be visible.

2. A case for Dstat

A few years ago I was involved in a project that was testing a storage cluster with a SAN back-end using GPFS and Samba for a broadcasting company. The performance tests that were scheduled together with the customer took a few weeks to measure the different behaviour under different stresses.

During these tests there was a need to see how each of the components behaved and to find problematic behaviour during testing. Also, because it involved 5 GPFS nodes, we needed to make sure that the load was spread evenly during the test. If everything went well repeatedly, the results were validated and the next batch of tests could be prepared and run.

We started off using different tools at first, but the more counters we were trying to capture the harder it was to post-process the information we had collected. What's more, we often saw only after performing tests that the data was not representative because the numbers didn't add up. Sometimes it was caused by the massive setup of clients that were autonomously stressing the cluster. On other occasions we noticed that the network was the culprit. All in all, we lost time because we could only validate the results by relating numbers after the tests were completed and not during the tests.

Complicating the matter was the fact that 5 different nodes were involved and using the normal command line tools like vmstat, iostat or ifstat (which only showed us a small part of what was happening) was problematic as each needed a different terminal. Besides, not all information was interesting.

Eventually Dstat was born, to make a dull task more enjoyable.

After the project was finished I was able to correlate system resources with network throughput, TCP information, Samba sessions, GPFS throughput, accumulated block device throughput, HBA throughput, all within a single interval.

3. Dstat characteristics

There are many ideas incorporated into Dstat by design, and this section serves to list all of them. Not all of them may appeal to the task you are doing, but the combination may make it an appealing proposition nevertheless.

3.1. History of counters

An important characteristic in tools like vmstat, iostat or ifstat is the fact that you can compare historical collected data with new data. This allows you to have a good feeling of how something is evolving.

Compare this to tools like top, htop or nmon, where data is often being refreshed and you loose historical information.

3.2. Adding unit indication

It was very important that when numbers were compared, they were in the same unit, and not eg. a different power exponent. The human mind sometimes works in mysterious ways and more so when working with numbers for hours and hours. Adding the unit is something very convenient and may reduce the human error factor.

Additionally, indicating the unit also makes sure that the columns have a fixed width. Often when using vmstat or other tools, the columns tend to shift depending on the width of the counter. This makes it very inconvenient to find counters in the shifted output.

3.3. Colour highlighting units

After I added colours to help improve indicating unites, I noticed that the colours also helped to show patterns. This of course is very limited, nevertheless it instantly shows when numbers are flat or changes are taking place.

IMPORTANT: The colours are arbitrarily chosen. Do not make the mistake to assume that green means good and red means bad. There is no real meaning to the colour itself, however a change of colour does mean that a value has gone over some pre-defined limit.

3.4. Intermediate updates

During tests, when you choose to see average values over a given time, it can be useful to see how the averages evolve. Dstat, by default, displays intermediate updates. This means that if you select to see 10 second averages, after each second you see the accumulated average over the timespan. **This means that after 4 seconds with intermediate updates, you see an average taken over the 4 second timeframe.**

NOTE: This means that the closer you get to the given timeframe (eg. 10 seconds) the more likely that it nears its final average over that period.

3.5. Adding custom counters

Dstat was specifically designed to enable anyone to add their own counters in a matter of minutes. The plugin-based system takes care of displaying, colouring and adding units to the counters. As a plugin-writer, you only have to focus on extracting the counters from the kernel (procfs or sysfs), logfiles or daemons.

3.6. Selecting plugins and counters

Being able to add custom counters is important, but selecting those counters that you really need is even more important if you want to correlate counters and see patterns. Less is more.

NOTE: In fact, Dstat currently does not allow you to select just counters, it only allows you to select plugins. However, since you can modify or fork a plugin, you still have the ability to select just those counters you prefer.

3.7. Exporting to CSV

Having information on screen is one thing, you most likely need some hard evidence later to make your case. (Why else do all the work?)

Dstat allows to write out all counters in the greatest detail possible to CSV. By default it also adds the command-line used for generating the output, as well as a date and time stamp. Since Dstat in the first place is meant for human-readable real-time statistics, it will by default also display the counters to screen (unless you */dev/null* it).

TIP: Dstat appends to the output file so that you can add tests-results of different tests to a single file. However, make sure that you tag each test properly (eg. by using distinct filenames for each different test).

3.8. Time-plugin included

It may seem a small thing, but having exact time (and date) information for your counters allows for a completely different usage as well. By adding simple date and time information, Dstat can be used as a background process in a screen to monitor the behaviour of your system during the night.

This proves to be very valuable for example to find offending processes during nightly tasks or to pinpoint their behaviour to certain events that you cannot monitor during working hours.

It is also important when you have multiple Dstats running (eg. for nodes in a cluster) to correlate counters between the outputs.

3.9. Terminal capabilities

Dstat also takes into account the width and height of your terminal window and modifies output to fit into your terminal. This, of course, has no effect on what ends up in the CSV output.

Another (debatable) useful feature is that Dstat will modify the terminal title to indicate on what system it was run and what options were used. Especially when monitoring nodes in a cluster, this can be useful, but even in Gnome finding your Dstat window is handy.

WARNING: Some people however are annoyed by the fact that their distribution does not reset the terminal title and Dstat therefor messes it up. There is no way for Dstat to fix this.

4. Plugins and counters

When we talk about plugins, we make a distinction between those plugins that are included within the Dstat tool itself, and those that ship with it externally. In essence there is no real difference, as the internal plugins could easily have been created as an external plugin. The basic difference is that the internal plugins have no dependencies except on procfs.

Having the basic plugins as part of Dstat, makes sure that Dstat can be moved as a self-contained file to other systems.

4.1. Internal plugins

The plugins that have been selected to be part of the Dstat tool itself, and therefor have no dependencies other than procfs, are:

- cpu, cpu24: CPU counters
- disk, disk24, disk24old: disk counters
- epoch: seconds since Epoch
- int, int24: interrupts per IRQ
- ipc: IPC counters
- load: load counters
- lock: locking counters
- mem: memory usage
- net: network usage
- page, page24: paging counters
- proc: process counters
- raw: raw socket counters
- swap, swapold: swap usage
- sys: system (kernel) counters
- tcp: TCP socket counters
- time: date and time
- udp: UDP socket counters
- unix: unix socket counters

For backward compatibility with older kernels there is a cascading system that selects the most appropriate internal plugin for your kernel. (eg. the `dstat_disk` plugin falls back to `dstat_disk24` and `dstat_disk24old`) At this moment there is no such system for external plugins.

4.2. External plugins

This basic functionality is easily extended by writing your own plugins (subclasses of the Dstat class) which are then inserted at runtime into Dstat. A set of *external* modules exist for:

- battery: battery usage
- cpufreq: CPU frequency
- dbus: DBUS connections
- freespace: free space on filesystems
- gpfsop: GPFS operations counters
- gpfs: GPFS IO counters
- innodb_io: innodb I/O counters
- innodb_keys: innodb key operation counters
- innodb_ops: innodb operations counters
- mysql_io: MySQL I/O counters
- mysql_ops: MySQL operations counters
- nfs3op: NFS3 client operations counters
- nfs3: NFS3 client counters
- nfsd3op: NFS3 server operations counters
- nfsd3: NFS3 server counters
- postfix: postfix queue counters
- rpcd: RPC server counters
- rpc: RPC client counters
- sendmail: sendmail queue counters
- thermal: thermal counters
- topbio: most expensive block I/O process
- topcpu: most expensive cpu process
- topio: most expensive I/O process
- topmem: most expensive memory process
- utmp: utmp counters
- vmkhba: VMware kernel HBA counters
- vmkint: VMWare kernel interrupt counters
- vzcpu: OpenVZ CPU counters
- vzubc: OpenVZ user beancounters
- wifi: WIFI quality information

4.3. Most-wanted plugins

Hoping someone interested reads this document, I added a few plugins that would be “very nice” to have but are currently lacking:

- slab: needs a VM expert to make sense out of the vast amount of data
- xorg: need information on how to get X resources, would be nice to see evolution of X resources over time
- samba: lacking information to get counters from Samba without forking smbstatus every second
- snmp: could be useful to relate counters from different systems in a single Dstat
- topx: display the most expensive X application(s)
- systemtap: connecting Dstat to systemtap counters

Creative souls with other ideas are welcome as well !

5. Using Dstat

Central to the Dstat command line interface is the selection of plugins. The selection and order of options influence the Dstat output directly.

5.1. Enabling plugins

The internal plugins have short and/or long options within Dstat, eg. `-c` or –`cpu` will enable the cpu counters.

The external plugins are enable by a single `-M` option followed by one or more plugin-names. (This also works for internal plugins)

The following examples will enable the time, cpu and disk plugins, and are equal.

```
dstat -tcd
dstat --time --cpu --disk
dstat -M time,cpu,disk
dstat -M time -M cpu -M disk
```

5.2. Total or individual counters

Some of the plugins can show both total values or individual values and therefor have an extra option to influence this decision.

```
dstat -d -D sda,sdb
dstat -n -N eth0,eth1
dstat -c -C total,0,1
```

You can show both the individual values and total values as follows:

```
[dag@horsea ~]$ dstat -d -D total,hda,hdc
-dsk/total----dsk/hda-----dsk/hdc--
 read  writ: read  writ: read  writ
1384k 1502k: 114k 1332k:  81k  359B
   0    44k:   0    44k:   0     0
   0     0 :   0     0 :   0     0
```

The special `-f` or –`full` option allows to select individual counters by default, and can be overruled by `-C`, `-D`, `-I`, `-N` or `-S`.

5.3. Influencing output

Dstat has a few more options to influence its output. With the –`nocolor` one can disable colours. The –`noheaders` option disables repeating headers. The –`noupdate` option disables intermediate updates. The –`output` option is used for writing out to a CSV file.

5.4. Plugin search path

Dstat looks in the following places for plugins. This allows a user without root privileges to use some extra plugins.

- ~/.dstat/
- ./plugins/
- .
- /usr/share/dstat/
- /usr/local/share/dstat/

The option `-M list` shows the available plugins and their location in the order that the plugin search path is used.

NOTE: Plugins are name `dstat_<name>.py`.

6. Use-cases

Below are some use-cases to demonstrate the usage of Dstat.

WARNING: The following examples do not look as nice as they do on screen because this document is not printed in colour (and I did not prepare it in colour :-)).

6.1. Simple system check

Let's say you quickly want to see if the system is doing alright. In the past this probably was a `vmstat 1`, as of now you would do:

```
dstat -taf
```

```
[dag@rhun dag]$ dstat -taf
-----time----- -------cpu0-usage------ --dsk/sda-----dsk/sr0-- --net/eth1- ---paging-- ---system--
  date/time   |usr sys idl wai hiq siq| read  writ: read  writ| recv  send|  in   out | int   csw
02-08 02:42:48| 10   2  85   2   0   0|  22k   23k: 1.8B    0 |   0     0 |2588B 2952B| 558   580
02-08 02:42:49|  4   3  93   0   0   0|   0     0 :   0     0 |   0     0 |   0     0 |1116   962
02-08 02:42:50|  5   2  90   0   2   1|   0    28k:   0     0 |   0     0 |   0     0 |1380  1136
02-08 02:42:51| 11   6  82   0   1   0|   0     0 :   0     0 |   0     0 |   0     0 |1277  1340
02-08 02:42:52|  3   3  93   0   1   0|   0    84k:   0     0 |   0     0 |   0     0 |1311  1034
```

NOTE: The `-t` here is completely optional and generally wastes space. But often you are not monitoring for 10 seconds but rather measure in minutes or hours. Having a general idea on what timescale counters have been averaged is nevertheless interesting.

6.2. What is this system doing now ?

I often run both the `dstat_topcpu` and `dstat_topmem` programs on a system, just to see what a system is doing. Having a quick look at what application is using the most CPU over a few minutes and to see what the general usage of memory is of the top application gives away a lot about a system.

```
[dag@horsea dag]$ dstat -c -M topcpu -dng -M topmem
----total-cpu-usage---- -most-expensive- -dsk/total- -net/total- ---paging-- -most-expensive-
usr sys idl wai hiq siq|  cpu process   | read  writ| recv  send|  in   out | memory process
  9   2  80   9   0   0|kswapd         0| 123k  164k|   0     0 |9196B  18k|rsync        74M
  2   3  95   0   0   0|sendmail       1|   0   168k|2584B  39k|   0     0 |rsync        74M
 18   3  79   0   0   0|httpd         17|   0    88k|5759B 118k|   0     0 |rsync        74M
  3   2  94   1   0   0|sendmail       1|4096B   0 |2291B 4190B|   0     0 |rsync        74M
  2   3  95   0   0   0|httpd          1|   0     0 |2871B 3201B|   0     0 |rsync        74M
 10   7  83   0   0   0|httpd         13|   0     0 |2216B  10k|   0     0 |rsync        74M
  2   2  96   0   0   0|                |   0    52k| 724B 2674B|   0     0 |rsync        74M
```

6.3. What process is using all my CPU or memory at 4:20 AM ?

Imagine the monitoring team notices strange peaks, a system engineer got a worthless message, the system was swaping extensively, a process got killed.

Something indicates the system is doing something unexpected but what is causing it and why ? As of now you can do:

```
dstat -tcy -M topcpu 120
dstat -tmgs -M topmem 120
dstat -td -M topio 120
```

to see what process is using the most CPU, the most memory and the most I/O resources.

And hopefully one day we can do:

```
dstat -tn -M topnet 120
dstat -tn -M topx 120
```

Leave it running during the night and in the morning you can see the light.

6.4. What device is slowing down my system ?

A nice feature of Dstat is that it can show how many interrupts each of your devices is generating. The *cpu* stats already show this in percentage as *hard interrupt* and *soft interrupt*, and the *sys* stats shows the total number of interrupts, but the *int* stats go into detail. And you can specify exactly what IRQs you want to watch.

Much like `watch -n1 -d cat /proc/interrupts` on speed.

```
dstat -t -y -i -f
```

which then results in:

```
[dag@rhun ~]$ dstat -t -y -i -f 5
-----time----- ---system-- -------------------interrupts-----------------
  date/time   | int   csw |  1     9     12    14    15    58   177   185
13-08 21:52:53| 740   923 |   1     0    18     5     1    17     4   131
13-08 21:52:58|1491  2085 |   0     4   351     1     2    37     0    97
13-08 21:53:03|1464  1981 |   0     0   332     1     3    31     0    96
13-08 21:53:08|1343  1977 |   0     0   215     1     2    32     0    93
13-08 21:53:13|1145  1918 |   0     0    12     0     3    33     0    95
```

When having the following hardware:

```
[dag@rhun ~]$ cat /proc/interrupts
           CPU0
  0:  143766685    IO-APIC-edge  timer
  1:     374043    IO-APIC-edge  i8042
  9:     102564   IO-APIC-level  acpi
 12:    4481057    IO-APIC-edge  i8042
 14:    1192508    IO-APIC-edge  libata
 15:     358891    IO-APIC-edge  libata
 58:    4391819   IO-APIC-level  ipw2200
177:     993740   IO-APIC-level  Intel ICH6
185:   33542364   IO-APIC-level  yenta, uhci_hcd:usb1, eth0, i915@pci:0000:00:02.0
NMI:          0
LOC:  143766578
ERR:          0
MIS:          0
```

Or select specific interrupts:

```
dstat -t -y -i -I 12,58,185 -f 5
```

6.5. How does my WIFI signal evolve when I move my laptop or AP through the house ?

Something I was looking into when trying to find the optimal location for the WIFI access point. However I must say that another tool I wrote *Dwscan* is currently more sophisticated.

```
dstat -t -M wifi
```

6.6. Is my SWRAID performing as it claims ?

This surprised me when Googling for Dstat. I was looking for other use-cases on the Internet and on the linux kernel mailinglist one of the SWRAID developers was indicating a problem with an implementation using Dstat output to prove it.

7. Writing your own Dstat plugin

Dstat is completely written in python and this makes it extremely convenient to write your own plugins. The many plugins that come with Dstat are an excellent source of information if you want to write your own.

7.1. Introducing the hello world plugin

The following plugin does nothing more than write "Hello world!" to its output.

```
class dstat_helloworld(dstat):
    def __init__(self):
        self.name = 'plugin title'
        self.format = ('s', 12, 100)
        self.nick = ('counter',)
        self.vars = ('text',)
        self.init(self.vars, 1)

    def extract(self):
        self.val['text'] = 'Hello world!'
```

In this example, there are several components:

1. `self.name` contains the plugin's visible title.
2. `self.format` is a list containing: the counter type, the counter length and how the colouring is done.
3. `self.nick` is a list of the counter names.
4. `self.vars` is a list of the variable names for each counter.
5. `self.init()` is a function that initialises the counter structures (`self.cn1`, `self.cn2` and `self.val`).
6. `self.val` contains the counter values that are being displayed.

7.2. Parsing counters

The following example shows how information is collected and counters are processed. It also includes a `check()` method to properly bail out when the system fails to meet some plugin criteria.

```
global glob
import glob

class dstat_postfix(dstat):
    def __init__(self):
        self.name = 'postfix'
        self.format = ('d', 4, 100)
        self.vars = ('incoming', 'active', 'deferred', 'bounce', 'defer')
        self.nick = ('inco', 'actv', 'dfrd', 'bnce', 'defr')
        self.init(self.vars, 1)

    def check(self):
        if not os.access('/var/spool/postfix/active', os.R_OK):
            raise Exception, 'Cannot access postfix queues'
        return True

    def extract(self):
        for item in self.vars:
            self.val[item] = len(glob.glob('/var/spool/postfix/'+item+'/*/*'))
```

This example shows the following items:

1. Since the plugin is imported at runtime, it is important that these are are included in the global scope to reuse them.
2. `self.format` indicates values are in decimal, counters are 4 characters wide and colouring differs every multiplication of 100.

3. The `check()` method tests conditions and bails out of they are not met.

4. To make processing easier we have opted to use as value names (`self.vars`) the name of the postfix queues and store counts in `self.val.`

7.3. Opening files

Dstat provides its own `dopen()` function to plugins. Using `dopen()` instead of `open()` plugins do not need to reopen files to update their counters. But this is only useful when plugins open a few files. For eg. opening */proc/pid* files the number of open files would only be increasing as the number of processes increases.

7.4. Piping to an application

Dstat provides its own `dpopen()` function to plugins. This function allows the plugin to open a stdin, stdout and stderr pipes for 2-way communication with processes. To see this in action, take a look at the `dstat_gpfs` plugins or the `dstat_mysql` plugins.

Piping to an application is more expensive than getting kernel counters from */proc*, but it beats having to run a program and capturing the output.

8. Known issues

There are some known issues that are important to understand when using Dstat.

8.1. Counter rollovers

Unfortunately Dstat is susceptible for counters that “rollover”. This means that a counter gets bigger than its maximum value the data-structure is capable of storing. As a result the counter is reset.

For some architectures and some counters, Linux implements 32bit values, this means that such counter can go up to 2^32 (= 4294967296B = 4G) values.

For example the network counters are calculated in absolute bytes. Every 4GB that is being transferred over the network will cause a counter reset. For example on a bonded 2x10Gbps interfaces that is using its theoretical transfer limit, this would happen every 1.6 seconds.

Since */proc* is updated every second, this would be impossible for Dstat to catch. Currently if Dstat encounters a negative difference for an interval it displays a dash.

Obviously, if Dstat were to know what the counter's maximum value is, it could recalculate the difference. However that is currently not implemented and does not guarantee a correct result either, since a negative value could be the result of 2 or more rollovers.

If you suspect that the behaviour of your system is susceptible of counter rollovers, make sure you take this into account when using Dstat (or any other tool that uses these counters for that matter)

TIP: Shipped with the Dstat documentation there is a document (*counter-rollovers.txt*) that goes deeper into counter rollovers. If this affects you, read that document and contact me for possible implementation changes to improve handling them.

8.2. Dstat performance

As mentioned several times now, Dstat is written in python. There are various reasons that Python was chosen and the most important reason is that it simplifies writing plugins, processing counters and lowers the bar for people to contribute changes.

The downside of choosing a scripting language is that it is slower than if it would be written in C, obviously. **Dstat is not optimised for performance.**

NOTE: This may seem ironic: a performance monitoring tool that is not optimised for performance, but rather for flexibility. However the ease of writing plugins and prototyping gets precedence over performance at this time.

8.2.1. Plugin performance

If we look at the basic plugins, there are no real performance issues with Dstat. Loading Dstat takes longer than eg. vmstat, but once running, Dstat's performance for the same functionality is up to par with vmstat, ifstat and other similar tools.

However there are plugins that are much more resource intensive than others and the selection of plugins determines Dstat's performance in a major way.

8.2.2. Debugging Dstat

Dstat comes with a –`debug` option that helps to find the cost of running plugins. The –`debug` option show how long it takes Dstat to process the selected plugins.

You can see the cost of Dstat itself by simply using the `dstat_time` plugin together with the –`debug` option.

```
[dag@rhun dag]$ dstat -t --debug
Module dstat_time
-----time-----
  date/time
19-08 20:34:21  5.90ms
19-08 20:34:22  0.17ms
19-08 20:34:23  0.18ms
19-08 20:34:24  0.18ms
```

Compare this with other plugins to see what the cost is of an individual plugin.

```
[dag@rhun dstat]$ dstat -c --debug
Module dstat_cpu requires ['/proc/stat']
----total-cpu-usage----
usr sys idl wai hiq siq
 15   3  77   4   0   1 11.07ms
  5   3  92   0   0   0  0.66ms
  5   4  91   0   0   0  0.65ms
  5   3  92   0   0   0  0.66ms
```

As you can see, getting the CPU counters and calculating the CPU usage takes up 0.5 milliseconds on this particular system. But if we look at the usage of the `dstat_topcpu` plugin:

```
[dag@rhun dstat]$ dstat -M topcpu --debug
Module dstat_topcpu
-most-expensive-
  cpu process
Xorg           2 43.82ms
Xorg           1 33.23ms
firefox-bin    2 33.54ms
Xorg           1 33.24ms
```

we see that processing the */proc/pid* files causes the topcpu plugin to use an additional 33ms.

WARNING: These values show the time it takes to process the plugins and does not indicate the amount of CPU usage Dstat consumes. This obviously means that the process time of plugins depends on how hard the system is being stressed as well as on what the plugin exactly is doing.

Plugins that communicate with other processes or those that process lots of information (for example communicating with the mysql client, or processing the mail queue) may not actually use any local resources, but the latency causes Dstat to slow down processing other counters.

8.2.3. Writing Dstat and plugins in C

It makes sense to reimplement Dstat or some of its plugins in C and still allow the writing of Python (or even Perl) plugins. Tests have shown that for example processing */proc/pid* in C makes the plugin 3 times faster. And this did not take into account the processing of the results and displaying the output.

So rewriting in C makes a lot of sense, but it is also much more complicated.

8.3. Python 1.5

Dstat works with python 2.0, however there is also a Dstat15 version that still works on python 1.5. The downside of having Dstat work on python 1.5 is that the external plugins cannot use the newer and more flexible python 2.0 syntax and the differences between python 1.5 and 2.0 are considerable.

NOTE: Not all plugins work properly on Python 1.5.

9. Future development

The Dstat release contains a *TODO* file highlighting all the items and ideas that have been played with. Here is a list of the most important ones:

- Output
  - Changes in how Dstat colours digits within a value (the 6 in 6134B) *
- Exporting information
  - Connecting Dstat with rrdtool

- Exporting to syslog or remote syslog (a way to transport counters ?)
- Plugins
  - Be smart when plugins are loaded more than once (some plugins could benefit)
  - Add more plugins
- Redesign Dstat
  - Work on the plugin infrastructure, make the API more simple and straightforward
  - Create an object-model and namespace for plugins and counters so that other tools can be based on Dstat

10. Links

- Dstat homepage
- Dstat subversion
- Dstat mailinglist