Monday, November 30, 2015

Linux Performance Analysis in 60,000 Milliseconds

You login to a Linux server with a performance issue: what do you check in the first minute?

At Netflix we have a massive EC2 Linux cloud, and numerous performance analysis tools to monitor and investigate its performance. These include Atlas for cloud-wide monitoring, and Vector for on-demand instance analysis. While those tools help us solve most issues, we sometimes need to login to an instance and run some standard Linux performance tools.

In this post, the Netflix Performance Engineering team will show you the first 60 seconds of an optimized performance investigation at the command line, using standard Linux tools you should have available.

First 60 Seconds: Summary

In 60 seconds you can get a high level idea of system resource usage and running processes by running the following ten commands. Look for errors and saturation metrics, as they are both easy to interpret, and then resource utilization. Saturation is where a resource has more load than it can handle, and can be exposed either as the length of a request queue, or time spent waiting.

```
uptime
dmesg | tail
vmstat 1
mpstat -P ALL 1
pidstat 1
iostat -xz 1
free -m
sar -n DEV 1
sar -n TCP,ETCP 1
top
```

Some of these commands require the sysstat package installed. The metrics these commands expose will help you complete some of the USE Method: a methodology for locating performance bottlenecks. This involves checking utilization, saturation, and error metrics for all resources (CPUs, memory, disks, e.t.c.). Also pay attention to when you have checked and exonerated a resource, as by process of elimination this narrows the targets to study, and directs any follow on investigation.

The following sections summarize these commands, with examples from a production system. For more information about these tools, see their man pages.

1. uptime

```
$ uptime
 23:51:26 up 21:31,  1 user,  load average: 30.02, 26.43, 19.02
```

This is a quick way to view the load averages, which indicate the number of tasks (processes) wanting to run. On Linux systems, these numbers include processes wanting to run on CPU, as well as processes blocked in uninterruptible I/O (usually disk I/O). This gives a high level idea of resource load (or demand), but can't be properly understood without other tools. Worth a quick look only.

The three numbers are exponentially damped moving sum averages with a 1 minute, 5 minute, and 15 minute constant. The three numbers give us some idea of how load is changing over time. For example, if you've been asked to check a problem server, and the 1 minute value is much lower than the 15 minute value, then you might have logged in too late and missed the issue.

In the example above, the load averages show a recent increase, hitting 30 for the 1 minute value, compared to 19 for the 15 minute value. That the numbers are this large means a lot of something: probably CPU demand; vmstat or mpstat will confirm, which are commands 3 and 4 in this sequence.

2. dmesg | tail

Links

Netflix US & Canada Blog
Netflix America Latina Blog
Netflix Brasil Blog
Netflix Benelux Blog
Netflix DACH Blog
Netflix France Blog
Netflix Nordics Blog
Netflix UK & Ireland Blog
Netflix ISP Speed Index
Open positions at Netflix
Netflix Website
Facebook Netflix Page
Netflix UI Engineering
RSS Feed

About the Netflix Tech Blog

This is a Netflix blog focused on technology and technology issues. We'll share our perspectives, decisions and challenges regarding the software we build and use to create the Netflix service.

Blog Archive

▼ 2015 (43)
▼ November (5)
Linux Performance Analysis in 60,000 Milliseconds
Creating Your Own EC2 Spot Market -- Part 2
Sleepy Puppy Extension for Burp Suite
Global Continuous Delivery with Spinnaker
Netflix Hack Day - Autumn 2015
► October (5)
► September (6)
► August (6)
► July (3)
► June (2)
► May (2)
► April (3)

```
$ dmesg | tail
[1880957.563150] perl invoked oom-killer: gfp_mask=0x280da, order=0, oom_score_adj=0
[...]
[1880957.563400] Out of memory: Kill process 18694 (perl) score 246 or sacrifice child
[1880957.563408] Killed process 18694 (perl) total-vm:1972392kB, anon-rss:1953348kB, file-r
ss:0kB
[2320864.954447] TCP: Possible SYN flooding on port 7001. Dropping request.  Check SNMP cou
nters.
```

This views the last 10 system messages, if there are any. Look for errors that can cause performance issues. The example above includes the oom-killer, and TCP dropping a request.

Don't miss this step! dmesg is always worth checking.

3. vmstat 1

```
$ vmstat 1
procs ---------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
34  0    0 200889792  73708 591828    0    0     0     5    6   10 96  1  3  0  0
32  0    0 200889920  73708 591860    0    0     0   592 13284 4282 98  1  1  0  0
32  0    0 200890112  73708 591860    0    0     0     0 9501 2154 99  1  0  0  0
32  0    0 200889568  73712 591856    0    0     0    48 11900 2459 99  0  0  0  0
32  0    0 200890208  73712 591860    0    0     0     0 15898 4840 98  1  1  0  0
^C
```

Short for virtual memory stat, vmstat(8) is a commonly available tool (first created for BSD decades ago). It prints a summary of key server statistics on each line.

vmstat was run with an argument of 1, to print one second summaries. The first line of output (in this version of vmstat) has some columns that show the average since boot, instead of the previous second. For now, skip the first line, unless you want to learn and remember which column is which.

Columns to check:

- **r**: Number of processes running on CPU and waiting for a turn. This provides a better signal than load averages for determining CPU saturation, as it does not include I/O. To interpret: an "r" value greater than the CPU count is saturation.
- **free**: Free memory in kilobytes. If there are too many digits to count, you have enough free memory. The "free -m" command, included as command 7, better explains the state of free memory.
- **si, so**: Swap-ins and swap-outs. If these are non-zero, you're out of memory.
- **us, sy, id, wa, st**: These are breakdowns of CPU time, on average across all CPUs. They are user time, system time (kernel), idle, wait I/O, and stolen time (by other guests, or with Xen, the guest's own isolated driver domain).

The CPU time breakdowns will confirm if the CPUs are busy, by adding user + system time. A constant degree of wait I/O points to a disk bottleneck; this is where the CPUs are idle, because tasks are blocked waiting for pending disk I/O. You can treat wait I/O as another form of CPU idle, one that gives a clue as to why they are idle.

System time is necessary for I/O processing. A high system time average, over 20%, can be interesting to explore further: perhaps the kernel is processing the I/O inefficiently.

In the above example, CPU time is almost entirely in user-level, pointing to application level usage instead. The CPUs are also well over 90% utilized on average. This isn't necessarily a problem; check for the degree of saturation using the "r" column.

4. mpstat -P ALL 1

```
$ mpstat -P ALL 1
Linux 3.13.0-49-generic (titanclusters-xxxxx)  07/14/2015  _x86_64_ (32 CPU)

07:38:49 PM  CPU   %usr  %nice   %sys %iowait   %irq  %soft  %steal  %guest  %gnice  %idle
07:38:50 PM  all  98.47   0.00   0.75    0.00   0.00   0.00    0.00    0.00    0.00   0.78
07:38:50 PM    0  96.04   0.00   2.97    0.00   0.00   0.00    0.00    0.00    0.00   0.99
07:38:50 PM    1  97.00   0.00   1.00    0.00   0.00   0.00    0.00    0.00    0.00   2.00
07:38:50 PM    2  98.00   0.00   1.00    0.00   0.00   0.00    0.00    0.00    0.00   1.00
07:38:50 PM    3  96.97   0.00   0.00    0.00   0.00   0.00    0.00    0.00    0.00   3.03
[...]
```

▶ March (3)
▶ February (5)
▶ January (3)

▶ 2014 (37)
▶ 2013 (52)
▶ 2012 (37)
▶ 2011 (17)
▶ 2010 (8)

Labels

accelerated compositing (2)
adwords (1)
Aegisthus (1)
algorithms (2)
aminator (1)
analytics (4)
Android (2)
angular (1)
api (16)
appender (1)
Archaius (2)
architectural design (1)
architecture (1)
Asgard (1)
Astyanax (3)
authentication (1)
automation (2)
autoscaling (3)
availability (4)
AWS (29)
benchmark (2)
big data (10)
billing (1)
Blitz4j (1)
build (3)
Cable (1)
caching (2)
Cassandra (13)
chaos engineering (1)
chaos monkey (5)
ci (1)
classloaders (1)
Clojure (1)
cloud (23)
cloud architecture (15)

This command prints CPU time breakdowns per CPU, which can be used to check for an imbalance. A single hot CPU can be evidence of a single-threaded application.

5. pidstat 1

```
$ pidstat 1
Linux 3.13.0-49-generic (titanclusters-xxxxx)  07/14/2015    _x86_64_    (32 CPU)

07:41:02 PM   UID       PID    %usr %system  %guest    %CPU   CPU  Command
07:41:03 PM     0         9    0.00    0.94    0.00    0.94     1  rcuos/0
07:41:03 PM     0      4214    5.66    5.66    0.00   11.32    15  mesos-slave
07:41:03 PM     0      4354    0.94    0.94    0.00    1.89     8  java
07:41:03 PM     0      6521 1596.23    1.89    0.00 1598.11    27  java
07:41:03 PM     0      6564 1571.70    7.55    0.00 1579.25    28  java
07:41:03 PM 60004     60154    0.94    4.72    0.00    5.66     9  pidstat

07:41:03 PM   UID       PID    %usr %system  %guest    %CPU   CPU  Command
07:41:04 PM     0      4214    6.00    2.00    0.00    8.00    15  mesos-slave
07:41:04 PM     0      6521 1590.00    1.00    0.00 1591.00    27  java
07:41:04 PM     0      6564 1573.00   10.00    0.00 1583.00    28  java
07:41:04 PM   108      6718    1.00    0.00    0.00    1.00     0  snmp-pass
07:41:04 PM 60004     60154    1.00    4.00    0.00    5.00     9  pidstat
^C
```

Pidstat is a little like top’s per-process summary, but prints a rolling summary instead of clearing the screen. This can be useful for watching patterns over time, and also recording what you saw (copy-n-paste) into a record of your investigation.

The above example identifies two java processes as responsible for consuming CPU. The %CPU column is the total across all CPUs; 1591% shows that that java processes is consuming almost 16 CPUs.

6. iostat -xz 1

```
$ iostat -xz 1
Linux 3.13.0-49-generic (titanclusters-xxxxx)  07/14/2015  _x86_64_ (32 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          73.96    0.00    3.73    0.03    0.06   22.21

Device:   rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await
xvda        0.00     0.23    0.21    0.18     4.52     2.08    34.37     0.00    9.98
xvdb        0.01     0.00    1.02    8.94   127.97   598.53   145.79     0.00    0.43
xvdc        0.01     0.00    1.02    8.86   127.79   595.94   146.50     0.00    0.45
dm-0        0.00     0.00    0.69    2.32    10.47    31.69    28.01     0.01    3.23
dm-1        0.00     0.00    0.00    0.94     0.01     3.78     8.00     0.33  345.84
dm-2        0.00     0.00    0.09    0.07     1.35     0.36    22.50     0.00    2.55
[...]
^C
```

This is a great tool for understanding block devices (disks), both the workload applied and the resulting performance. Look for:

- **r/s, w/s, rkB/s, wkB/s**: These are the delivered reads, writes, read Kbytes, and write Kbytes per second to the device. Use these for workload characterization. A performance problem may simply be due to an excessive load applied.
- **await**: The average time for the I/O in milliseconds. This is the time that the application suffers, as it includes both time queued and time being serviced. Larger than expected average times can be an indicator of device saturation, or device problems.
- **avgqu-sz**: The average number of requests issued to the device. Values greater than 1 can be evidence of saturation (although devices can typically operate on requests in parallel, especially virtual devices which front multiple back-end disks.)
- **%util**: Device utilization. This is really a busy percent, showing the time each second that the device was doing work. Values greater than 60% typically lead to poor performance (which should be seen in await), although it depends on the device. Values close to 100% usually indicate saturation.

If the storage device is a logical disk device fronting many back-end disks, then 100% utilization may just mean that some I/O is being processed 100% of the time, however, the back-end disks may be far from saturated, and may be able to handle much more work.

Bear in mind that poor performing disk I/O isn’t necessarily an application issue. Many techniques are

cloud prize (3)
CO2 (1)
collection (1)
computer vision (1)
concurrency (1)
configuration (2)
configuration management (2)
conformity monkey (1)
content platform engineering (2)
continuous delivery (4)
coordination (2)
cost management (1)
crypto (1)
Cryptography (2)
CSS (2)
CUDA (1)
dart (1)
database (4)
data migration (1)
data pipeline (4)
data science (6)
DataStax (2)
data visualization (1)
deadlock (1)
deep learning (1)
Denominator (2)
dependency injection (1)
device (3)
device proliferation (1)
devops (2)
distributed (10)
DNS (1)
Docker (1)
Dockerhub (1)
DSL (1)
Dyn (1)
DynECT (1)
efficiency (1)
Elastic Load Balancer (1)
elasticsearch (1)
ELB (1)
EMR (2)
encoding (2)
energy (1)
eucalyptus (1)

typically used to perform I/O asynchronously, so that the application doesn't block and suffer the latency directly (e.g., read-ahead for reads, and buffering for writes).

7. free -m

```
$ free -m
             total       used       free     shared    buffers     cached
Mem:        245998      24545     221453         83         59        541
-/+ buffers/cache:      23944     222053
Swap:            0          0          0
```

The right two columns show:

- **buffers**: For the buffer cache, used for block device I/O.
- **cached**: For the page cache, used by file systems.

We just want to check that these aren't near-zero in size, which can lead to higher disk I/O (confirm using iostat), and worse performance. The above example looks fine, with many Mbytes in each.

The "-/+ buffers/cache" provides less confusing values for used and free memory. Linux uses free memory for the caches, but can reclaim it quickly if applications need it. So in a way the cached memory should be included in the free memory column, which this line does. There's even a website, linuxatemyram, about this confusion.

It can be additionally confusing if ZFS on Linux is used, as we do for some services, as ZFS has its own file system cache that isn't reflected properly by the free -m columns. It can appear that the system is low on free memory, when that memory is in fact available for use from the ZFS cache as needed.

8. sar -n DEV 1

```
$ sar -n DEV 1
Linux 3.13.0-49-generic (titanclusters-xxxxx)  07/14/2015     _x86_64_    (32 CPU)

12:16:48 AM     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxm
12:16:49 AM      eth0  18763.00   5032.00  20686.42    478.30      0.00      0.00
12:16:49 AM        lo     14.00     14.00      1.36      1.36      0.00      0.00
12:16:49 AM   docker0      0.00      0.00      0.00      0.00      0.00      0.00

12:16:49 AM     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxm
12:16:50 AM      eth0  19763.00   5101.00  21999.10    482.56      0.00      0.00
12:16:50 AM        lo     20.00     20.00      3.25      3.25      0.00      0.00
12:16:50 AM   docker0      0.00      0.00      0.00      0.00      0.00      0.00
^C
```

Use this tool to check network interface throughput: rxkB/s and txkB/s, as a measure of workload, and also to check if any limit has been reached. In the above example, eth0 receive is reaching 22 Mbytes/s, which is 176 Mbits/sec (well under, say, a 1 Gbit/sec limit).

This version also has %ifutil for device utilization (max of both directions for full duplex), which is something we also use Brendan's nicstat tool to measure. And like with nicstat, this is hard to get right, and seems to not be working in this example (0.00).

9. sar -n TCP,ETCP 1

```
$ sar -n TCP,ETCP 1
Linux 3.13.0-49-generic (titanclusters-xxxxx)  07/14/2015    _x86_64_    (32 CPU)

12:17:19 AM  active/s passive/s    iseg/s    oseg/s
12:17:20 AM      1.00      0.00  10233.00  18846.00

12:17:19 AM  atmptf/s  estres/s retrans/s isegerr/s   orsts/s
12:17:20 AM      0.00      0.00      0.00      0.00      0.00

12:17:20 AM  active/s passive/s    iseg/s    oseg/s
12:17:21 AM      1.00      0.00   8359.00   6039.00

12:17:20 AM  atmptf/s  estres/s retrans/s isegerr/s   orsts/s
12:17:21 AM      0.00      0.00      0.00      0.00      0.00
^C
```

This is a summarized view of some key TCP metrics. These include:

eureka (2)
evcache (1)
failover (2)
falcor (2)
fault-tolerance (12)
flamegraphs (1)
Flow (1)
footprint (1)
FRP (1)
functional reactive (1)
garbage (1)
garbage collection (1)
gc (1)
Genie (4)
Governator (1)
GPU (2)
green (1)
Groovy (1)
Hack Day (3)
Hadoop (12)
HBase (1)
high volume (4)
high volume distributed systems (8)
Hive (2)
HTML5 (7)
https (1)
Hystrix (5)
IBM (1)
ice (1)
initialization (1)
innovation (3)
insights (1)
inter process communication (1)
Ipv6 (2)
isolation (1)
ISP (1)
java (5)
JavaScript (17)
jclouds (1)
jenkins (1)
kafka (1)
Karyon (2)
lifecycle (1)
linux (2)
lipstick (2)

- **active/s**: Number of locally-initiated TCP connections per second (e.g., via connect()).
- **passive/s**: Number of remotely-initiated TCP connections per second (e.g., via accept()).
- **retrans/s**: Number of TCP retransmits per second.

The active and passive counts are often useful as a rough measure of server load: number of new accepted connections (passive), and number of downstream connections (active). It might help to think of active as outbound, and passive as inbound, but this isn't strictly true (e.g., consider a localhost to localhost connection).

Retransmits are a sign of a network or server issue; it may be an unreliable network (e.g., the public Internet), or it may be due a server being overloaded and dropping packets. The example above shows just one new TCP connection per-second.

10. top

```
$ top
top - 00:15:40 up 21:56,  1 user,  load average: 31.09, 29.87, 29.92
Tasks: 871 total,   1 running, 868 sleeping,   0 stopped,   2 zombie
%Cpu(s): 96.8 us,  0.4 sy,  0.0 ni,  2.7 id,  0.1 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  25190241+total, 24921688 used, 22698073+free,    60448 buffers
KiB Swap:        0 total,        0 used,        0 free.   554208 cached Mem

   PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 20248 root      20   0  0.227t 0.012t  18748 S  3090  5.2  29812:58 java
  4213 root      20   0 2722544  64640  44232 S  23.5  0.0 233:35.37 mesos-slave
 66128 titancl+  20   0   24344   2332   1172 R   1.0  0.0   0:00.07 top
  5235 root      20   0 38.227g 547004  49996 S   0.7  0.2   2:02.74 java
  4299 root      20   0 20.015g 2.682g  16836 S   0.3  1.1  33:14.42 java
     1 root      20   0   33620   2920   1496 S   0.0  0.0   0:03.82 init
     2 root      20   0       0      0      0 S   0.0  0.0   0:00.02 kthreadd
     3 root      20   0       0      0      0 S   0.0  0.0   0:05.35 ksoftirqd/0
     5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
     6 root      20   0       0      0      0 S   0.0  0.0   0:06.94 kworker/u256:0
     8 root      20   0       0      0      0 S   0.0  0.0   2:38.05 rcu_sched
```

The top command includes many of the metrics we checked earlier. It can be handy to run it to see if anything looks wildly different from the earlier commands, which would indicate that load is variable.

A downside to top is that it is harder to see patterns over time, which may be more clear in tools like vmstat and pidstat, which provide rolling output. Evidence of intermittent issues can also be lost if you don't pause the output quick enough (Ctrl-S to pause, Ctrl-Q to continue), and the screen clears.

Follow-on Analysis

There are many more commands and methodologies you can apply to drill deeper. See Brendan's Linux Performance Tools tutorial from Velocity 2015, which works through over 40 commands, covering observability, benchmarking, tuning, static performance tuning, profiling, and tracing.

Tackling system reliability and performance problems at web scale is one of our passions. If you would like to join us in tackling these kinds of challenges we are hiring!

Posted by Brendan Gregg at 1:38 PM

G+1 +67 Recommend this on Google

Labels: linux, performance

Home   Older Post

load balancing (3)
localization (1)
localization platform engineering (1)
locking (1)
locks (1)
log4j (1)
logging (2)
machine learning (5)
Map-Reduce (1)
meetup (3)
memcache (2)
memcached (1)
message security layer (1)
Mobile (1)
modules (1)
monitoring (1)
msl (1)
negative keywords (1)
Netflix (16)
Netflix API (7)
netflix graph (1)
Netflix OSS (12)
neural networks (1)
node.js (3)
NoSQL (5)
observability (1)
Open source (10)
operational excellence (1)
operational insight (2)
operational visibility (1)
optimization (1)
OSS (3)
outage (1)
page generation (1)
payments (1)
Paypal (1)
performance (19)
personalization (4)
phone (1)
Pig (4)
pki (1)
Playback (1)
prediction (2)
predictive modeling (2)

Presto (2)

prize (1)

pubsub (1)

pytheas (1)

python (3)

Quality (1)

rca (2)

React (1)

Reactive Programming (2)

real-time insights (1)

real-time streaming (1)

Recipe (1)

recommendations (6)

Redis (2)

reinvent (2)

reliability (7)

remote procedure calls (1)

renewable (1)

research (1)

resiliency (7)

REST (2)

Ribbon (2)

Riot Games (1)

root-cause analysis (2)

Route53 (1)

rule engine (1)

Rx (1)

scalability (11)

scale (1)

scripting library (1)

search (2)

security (8)

Servo (1)

shared libraries (1)

simian army (5)

SimpleDB (3)

site reliability (1)

sqoop (1)

ssd (1)

ssl (2)

STAASH (1)

streaming (1)

stream processing (1)

suro (1)

SWF (1)

synchronization (1)

tablet (1)

testability (1)

tls (2)

traffic optimization (1)

TV (5)

UI (13)

UltraDNS (1)

unit test (2)

uptime (2)

user interface (5)

Velocity (1)

visualization (1)

WebKit (3)

websockets (1)

Wii U (1)

winner (1)

winners (1)

workflow (1)

ZeroToDocker (1)

ZooKeeper (1)

zuul (1)

“cloud architecture” (3)