SystemTap : observabilité intégrale des systèmes Linux

Adrien Kunysz
adk@acunu.com

Solutions Linux / Open Source
Paris, le 12 mai 2011

Qui est Adrien Kunysz ?

- Test Engineer chez Acunu Ltd
  - cloud storage / Big Data : Cassandra en plus performant
- j'aime regarder des cores, lire du code, analyser les problèmes de performance, bidouiller le noyau, la libc, les débogueurs,. . .
- co-fondateur de FSUGAr (Belgique)
- Krunch sur Freenode
- je suis juste un utilisateur de SystemTap (pas un développeur)

De quoi allons nous parler ?

Explication de SystemTap

Exemples pratiques

Prérequis et sécurité

Comparaison à d'autres outils

Conclusion

Bonus ?

Qu'est-ce que SystemTap ?

D'après http ://sourceware.org/systemtap/

> *SystemTap provides free software (GPL) infrastructure to simplify the gathering of information about the running Linux system. This assists diagnosis of a performance or functional problem. SystemTap eliminates the need for the developer to go through the tedious and disruptive instrument, recompile, install, and reboot sequence that may be otherwise required to collect data.*

- *framework* d'injection de code pour tout le système
- facilités pour les tâches de *tracing/debugging* courantes
- permet d'observer très facilement n'importe quoi dans un système en fonctionnement
  - . . . le problème est de savoir ce qu'on veut observer

Comment ça marche ?

1. écrire ou choisir un script décrivant ce qu'on veut observer
2. `stap` transforme le script en un module noyau
3. `stap` charge le module et communique avec lui
4. plus qu'à attendre les données

Les cinq passes de stap

```
# stap -v test.stp
Pass 1: parsed user script and 38 library script(s) in
    150usr/20sys/183real ms.
Pass 2: analyzed script: 1 probe(s), 5 function(s), 14
    embed(s), 0 global(s) in 110usr/110sys/242real ms.
Pass 3: translated to C into
    "/tmp/stapEjEd0T/stap_6455011c477a19ec8c7bbd5ac12a9cd0_13
     in 0usr/0sys/0real ms.
Pass 4: compiled C into
    "stap_6455011c477a19ec8c7bbd5ac12a9cd0_13608.ko" in
    1250usr/240sys/1685real ms.
Pass 5: starting run.
[...la sortie du script vient ici...]
Pass 5: run completed in 20usr/30sys/4204real ms.
```

Quelques exemples de *probe points* SystemTap

SystemTap exécute des *actions*
quand le code passe sur des *probe points*

- `syscall.read`
  - en entrant dans l'appel système `read()`
- `syscall.close.return`
  - en sortant de l'appel système `close()`
- `module("floppy").function("*")`
  - en entrant dans n'importe quelle fonction du module "floppy"
- `kernel.function("*@net/socket.c").return`
  - en sortant de n'importe quelle fonction dans le fichier `net/socket.c`
- `kernel.statement("*@kernel/sched.c :2917")`
  - en passant sur la ligne 2917 du fichier `kernel/sched.c`
- `timer.ms(200)`
  - toutes les 200 millisecondes
- `kernel.trace("mm_kswapd_runs")`
  - quand la VM a libéré des pages

Plus d'exemples de *probe points*

- `process("/bin/ls").function("*")`
  - en entrant dans n'importe quelle fonction de `/bin/ls` (pas ses bibliothèques ou appels systèmes)
- `process("/lib/libc.so.6").function("*malloc*")`
  - en entrant dans n'importe quelle fonction de la glibc qui a "malloc" dans son nom
- `process("postgres").mark("query__start")`
  - en démarrant une requête PostgreSQL
- `hotspot.method_entry`
  - en entrant dans n'importe quelle méthode Java
- `python.function.return`
  - en sortant de n'importe quelle fonction Python
- `kernel.function("*init*")`, `kernel.function("*exit*").return`
  - en entrant dans n'importe quelle fonction noyau qui a "init" dans son nom ou en sortant de n'importe quelle fonction qui a "exit" dans son nom

RTFM pour en savoir plus (`man stapprobes`).

Le langage de programmation SystemTap

- style C avec un esprit de `awk`
- tableaux associatifs
- agrégats de données statistiques
  - très facile de récolter des données et de faire des stats dessus (moyenne, min, max,...)
- beaucoup des fonctions disponibles (*builtin* et dans les *tapsets*)

RTFM : *SystemTap Language Reference* distribué avec SystemTap (langref.pdf)

Quelques fonctions que vous allez voir souvent

`pid()` de quel processus s'agit-il ?

`uid()` quel utilisateur fait tourner ce code ?

`execname()` quel est le nom de ce processus ?

`tid()` de quel *thread* s'agit-il ?

`gettimeofday_s()` temps *epoch* en secondes

`probefunc()` dans quelle fonction sommes-nous ?

`print_backtrace()` comment est-on arrivé jusqu'ici ?

Il y en a beaucoup d'autres. RTFM (`man stapfuncs`) et explorez `/usr/share/systemtap/tapset/`.

Quelques options utiles de `stap`

- `-x` tracer uniquement le PID spécifié
- `-c` lance la commande et ne trace qu'elle et ses enfants
- `-L` liste les *probe points* correspondants à l'argument ainsi que les variables disponibles à cet endroit
- `-g` mode guru : permet de modifier le comportement du code « observé »
  - potentiellement dangereux mais amusant et parfois utile

Agenda

Exemple 1 : tracer l'exécution des processus

Listing 1 – exec.stp

```
1 probe syscall.exec* {
2     printf("exec %s %s\n", execname(), argstr)
3 }
```

```
$ stap -L 'syscall.exec*'
syscall.execve name:string filename:string
   args:string argstr:string

# stap exec.stp
exec gnome-terminal /bin/bash
exec bash /usr/bin/id -gn
exec bash /usr/bin/id -un
exec bash /bin/uname -s
exec bash /bin/uname -r
```

Exemple 2 : un cas de support réel

Client Bonjour, le service `saslauthd` s'arrête mystérieusement de temps à autres. Est-ce que vous pouvez m'aider ?

Support OK, qu'est-ce que `strace` raconte ?

Client Il se prend un `SIGKILL`.

Support OK, essayons de déterminer qui envoi le signal.

Exemple 2 (suite) : sigkill.stp

Listing 2 – examples/process/sigkill.stp

```
1 # Copyright (C) 2007 Red Hat, Inc., Eugene Teo
2 [...GPL blah...]
3 probe signal.send {
4   if (sig_name == "SIGKILL")
5     printf("%s was sent to %s (pid:%d) by %s uid:%d\n",
6            sig_name, pid_name, sig_pid, execname(), uid())
7 }
```

```
$ stap -L signal.send
signal.send name:string sig:long task:long
   sinfo:long shared:long send2queue:long
   sig_name:string sig_pid:long pid_name:string
    si_code:string $sig:int

# stap /usr/share/systemtap/tapset/signal.stp
SIGKILL was sent to saslauthd (pid:6202) by
   AntiCloseWait.s uid:0
```

Exemple 2 (suite) : correction

```
$ find sosreport/ -name AntiCloseWait.s*
sosreport/etc/cron.hourly/AntiCloseWait.sh
```

Support Corrigez votre tâche cron.

Client Merci.

Exemple 3 : surveillance d'écriture/lecture de fichier

Listing 3 – filewatch.stp

```
1  probe kernel.function("vfs_write"),
       kernel.function("vfs_read")
2  {
3      dev_nr   = $file->f_path->dentry->d_inode->i_sb->s_dev
4      inode_nr = $file->f_path->dentry->d_inode->i_ino
5
6      if (dev_nr == stat2dev($1) && inode_nr == $2)
7          printf("%s(%d) %s 0x%x/%u\n",
8                 execname(), pid(), probefunc(),
9                 dev_nr, inode_nr)
10 }
11
12 # convert "stat -c %d" output to a proper device number
13 function stat2dev(s)
14 {
15     return ((s & 0xff00) << 12) | (s & 0xff)
16 }
```

Exemple 3 (suite) : utilisation de filewatch.stp

```
# stat -c 'device: %d, inode: %i' /etc/passwd
device: 64768, inode: 1805253
# stap filewatch.stp 64768 1805253
bash(28549) vfs_read 0xfd00000/1805253
id(28553) vfs_read 0xfd00000/1805253
crontab(28579) vfs_read 0xfd00000/1805253
id(28585) vfs_read 0xfd00000/1805253
vim(28620) vfs_read 0xfd00000/1805253
```

Agenda

Prérequis

- `CONFIG_KPROBES=y` (uniquement pour le *tracing* noyau)
- les *probes* se basant sur le code source nécessitent les symboles de déboguage
  - `paquet-debuginfo` sur les distros RPM
  - `packet-dbg` sur les distros .deb
  - compilez vos applications avec `gcc -g`
  - pour le noyau, c'est `CONFIG_DEBUG_INFO=y`
- pour le *tracing* en espace utilisateur, il faut le patch noyau *utrace*
  - pas (encore ?) dans la branche principale
  - intégré à Red Hat Enterprise Linux 5+, Fedora, CentOS,...
- pour les langages plus haut niveau, il faut un *runtime* instrumenté

Performances et sécurité

- sécurités au niveau du langage
  - pas de pointeurs
  - pas de boucles infinies
  - inférence de type
  - vous pouvez aussi écrire vos *probe handlers* en C (avec -g)
    à vos risques et périls
- sécurités au niveau du *runtime*
  - temps d'exécution max pour chaque *probe handler*
  - accès concurrents
  - *overload processing* (ne laisse pas `stap` prendre tout le CPU)
  - la plupart des sécurités peuvent être court-circuitées
    si nécessaire
  - cf la section `SAFETY AND SECURITY` de `stap(1)`

Les performances dépendent beaucoup de ce que vous faites mais, en général, SystemTap essaie de vous empêcher de faire n'importe quoi (mais on peut toujours le forcer si on veut vraiment).

Agenda

SystemTap vs DTrace

SystemTap est souvent décrit comme « DTrace pour Linux ». Je n'ai jamais utilisé DTrace mais de ce que j'en entend...

- fonctionnellement très proches
- langage différent
- implémentation très différent (machine virtuelle vs code natif)
- SystemTap est uniquement pour Linux (fonctionne *out of the box* au moins sur Debian, RHEL, CentOS, Fedora)
- DTrace existe sous Solaris, FreeBSD, OS X (et Linux ?)

SystemTap vs *auditd(8)*

- *auditd* ne peut tracer que les appels systèmes
- *auditd* ne peut pas vraiment faire de décodage/filtrage des arguments des appels systèmes
- selon la distro, *auditd* peut être plus facile à mettre en place

L'exemple sigkill.stp avec *auditd* :

```
# auditctl -a entry,always -S kill -F a1=9
```

Et ce qu'on trouve dans les logs quand on tue `sleep(1)` :

```
type=SYSCALL msg=audit(1275595476.234:430): arch=40000003
    syscall=37 success=yes exit=0 a0=5b25 a1=9 a2=5b25
    a3=5b25 items=0 ppid=23188 pid=23189 auid=500 uid=500
    gid=500 euid=500 suid=500 fsuid=500 egid=500 sgid=500
    fsgid=500 tty=pts2 ses=35 comm="bash" exe="/bin/bash"
    subj=user_u:system_r:unconfined_t:s0 key=(null)
type=OBJ_PID msg=audit(1275595476.234:430): opid=23333
    oauid=500 ouid=500 oses=35
    obj=user_u:system_r:unconfined_t:s0 ocomm="sleep"
```

SystemTap vs OProfile

OProfile récolte des échantillons tous les `$N` cycles CPU
pour essayer de déterminer ce sur quoi le CPU passe son temps.

- uniquement profilage d'utilisation CPU
- ne permet pas d'effectuer des actions complexe
  lors de l'échantillonnage
- fonctionne avec le noyau de base,
  même pour le profilage *userland*
- ne fonctionne pas dans la plupart des machines virtuelles

SystemTap vs outils en espace utilisateur

- *strace* ne gère que les appels systèmes
- *ltrace* ne gère que les fonctions *userland*
- $\{s,l\}$*trace* ne peut surveiller que des processus spécifiques
- $\{s,l\}$*trace* ne peut pas traiter les traces à la volée (statistiques, filtrage avancé,. . .)
- *gdb* est plutôt destiné au déboguage interactif

Références et questions

- wiki SystemTap : http ://sourceware.org/systemtap/wiki
- plein de documentation excellente fournie avec la distribution
  - `man -k stap`
  - file :///usr/share/doc/systemtap*
- il y a probablement déjà un script pour faire ce que vous voulez : http ://sourceware.org/systemtap/examples/
- systemtap@sources.redhat.com
- irc ://chat.freenode.net/#systemtap
- Big Data ? http ://acunu.com/

Agenda

Example 4 : callgraph for anything

Listing 4 – examples/general/para-callgraph.stp (simplified a bit)

```
1  function trace(entry_p, extra) {
2          printf("%s%s%s %s\n",
3                  thread_indent (entry_p),
4                  (entry_p>0?"->":"<-"),
5                  probefunc (),
6                  extra)
7  }
8
9  probe $1.call   { trace(1, $$parms) }
10 probe $1.return { trace(-1, $$return) }
```

Example 4 : using para-callgraph.stp

```
# stap examples/general/para-callgraph.stp
    'process("/usr/sbin/sendmail").function("*")'
     0 sendmail(4523):->doqueuerun
  1736 sendmail(4523):<-doqueuerun return=0x0
     0 sendmail(4523):->sm_blocksignal sig=0xe
    56 sendmail(4523):<-sm_blocksignal return=0x0
     0 sendmail(4523):->curtime
    22 sendmail(4523):<-curtime return=0x4c06fb34
     0 sendmail(4523):->refuseconnections name=0x93ad8b0
         e=0x343a80 d=0x0 active=0x0
    59 sendmail(4523): ->sm_getla
   109 sendmail(4523):  ->getla
   930 sendmail(4523):   ->sm_io_open type=0x3432c0
       timeout=0xfffffffffffffffe info=0x3231cd flags=0x2
       rpool=0x0
  1733 sendmail(4523):    ->sm_flags flags=0x2
  1771 sendmail(4523):    <-sm_flags return=0x10
  1876 sendmail(4523):    ->sm_fp t=0x3432c0 flags=0x10
      oldfp=0x0
 12409 sendmail(4523):    <-sm_fp return=0x372d7c
```

Example 5 : block I/O requests monitoring

Listing 5 – examples/io/ioblktime.stp (part 1 of 2)

```
1  global req_time, etimes
2
3  probe ioblock.request {
4    req_time[$bio] = gettimeofday_us()
5  }
6
7  probe ioblock.end {
8    t = gettimeofday_us()
9    s =  req_time[$bio]
10   delete req_time[$bio]
11   if (s) {
12     etimes[devname, bio_rw_str(rw)] <<< t - s
13   }
14 }
```

This is just to collect the data (no printing).

Example 5 continued : printing the collected data

Listing 6 – continuation of examples/io/ioblktime.stap (part 2 of 2)

```
15 probe timer.s(10), end {
16   ansi_clear_screen()
17   printf("%10s %3s %10s %10s %10s\n",
18          "device", "rw", "total (us)", "count", "avg (us)")
19   foreach ([dev,rw] in etimes- limit 20) {
20     printf("%10s %3s %10d %10d %10d\n", dev, rw,
21            @sum(etimes[dev,rw]), @count(etimes[dev,rw]),
22            @avg(etimes[dev,rw]))
23   }
24   delete etimes
25 }
```

Example 5 continued : what it looks like

```
# stap examples/io/ioblktime.stp
 device   rw total (us)        count    avg (us)
    sda    W  270301266         2160      125139
   dm-0    W  270344450         2160      125159
   dm-0    R      30010            4        7502
    sda    R      28615            4        7153
```

Requirements : instrumented language runtime

The lange runtime (VM, interpreter,. . .) needs to give hints to SystemTap as to what's going on in the upper layers.

- not required if you want to `stap` the interpreter/VM internals
- some languages already have DTrace probes
  - SystemTap can reuse them
  - this requires rebuilding the runtime
- AFAIK, only Fedora and RHEL6 (and derived) ship with instrumented language runtimes for now : Java (OpenJDK/IcedTea), Python, TCL, Perl ?
- check build options and bug database of your favourite language

Example 6 : watching Java threads

```
$ stap -l hotspot.thread*
hotspot.thread_start
hotspot.thread_stop
$ stap -L hotspot.thread*
hotspot.thread_start name:string thread_name:string
    id:long native_id:long is_daemon:long
    probestr:string $arg1:char const* volatile
    $arg2:int volatile $arg3:jlong volatile
    $arg4:pid_t volatile $arg5:bool volatile
hotspot.thread_stop name:string thread_name:string
    id:long native_id:long is_daemon:long
    probestr:string $arg1:char const* volatile
    $arg2:int volatile $arg3:jlong volatile
    $arg4:pid_t volatile $arg5:bool volatile
```

Example 6 continued : watching Java threads

Listing 7 – jthreads.stp

```
1  probe hotspot.thread_start {
2          printf("starting %s (%d)\n", thread_name, id)
3  }
4
5  probe hotspot.thread_stop {
6          printf("stopping %s (%d)\n", thread_name, id)
7  }
```

```
$ java -XX:+ExtendedDTraceProbes MyJavaApp
```

```
# stap jthreads.stp
starting Reference Handler (2)
starting Finalizer (3)
[...]
starting Lookup Dispatch Thread (34)
starting Folder Instance Processor (35)
starting Importer (36)
stopping Image Fetcher 0 (27)
starting Image Fetcher 0 (37)
[...]
```

Example 7 : call graph of a Python application

```
$ stap -L python.function.*
python.function.entry filename:string funcname:string
    lineno:long $arg1:char* volatile $arg2:char* volatile
    $arg3:int volatile
python.function.return filename:string funcname:string
    lineno:long $arg1:char* volatile $arg2:char* volatile
    $arg3:int volatile
```

Example 7 continued : call graph of a Python application

Listing 8 – python-callgraph.stp

```
1  function trace(entry_p, funcname, filename, lineno) {
2          if (isinstr(filename, @1)) {
3                  printf("%s%s%s@%s:%d\n",
4                          thread_indent(entry_p),
5                          (entry_p > 0 ? "->" : "<-" ),
6                          funcname, filename, lineno)
7          }
8  }
9
10 probe python.function.entry {
11         trace(1, funcname, filename, lineno)
12 }
13
14 probe python.function.return {
15         trace(-1, funcname, filename, lineno)
16 }
```

Example 7 continued : call graph of a Python application

```
# stap python-callgraph.stp yum
[...]
173181 yum(4458):    ->BaseConfig@yum/config.py:462
173202 yum(4458):    <-BaseConfig@yum/config.py:577
173240 yum(4458):    ->StartupConf@yum/config.py:586
173259 yum(4458):     ->__init__@yum/config.py:243
173275 yum(4458):      ->__init__@yum/config.py:58
173293 yum(4458):
    ->_setattrname@yum/config.py:63
173318 yum(4458):
    <-_setattrname@yum/config.py:67
173341 yum(4458):      <-__init__@yum/config.py:61
173353 yum(4458):     <-__init__@yum/config.py:246
[...]
173943 yum(4458):    <-StartupConf@yum/config.py:605
173982 yum(4458):    ->YumConf@yum/config.py:607
[...]
```

Example 8 : instrumenting PostgreSQL

How can I tell what queries are killing my PostgreSQL database right now ?

```
$ stap -L 'process("postgres").mark("query*")'
process("postgres").mark("query__done") $arg1:char
    const* volatile
process("postgres").mark("query__execute__done")
process("postgres").mark("query__execute__start")
process("postgres").mark("query__start") $arg1:char
     const* volatile
[...]
```

Example 8 continued : pgtop.stp

Listing 9 – pgtop.stp (part 1 : collecting the data)

```
1  global livequeries
2  global completequeries
3
4  probe process("postgres").mark("query__start") {
5          livequeries[tid(), $arg1] = gettimeofday_us()
6  }
7
8  probe process("postgres").mark("query__done") {
9          now = gettimeofday_us()
10         t = tid()
11         if ([t, $arg1] in livequeries) {
12                 delta = now - livequeries[t, $arg1]
13                 query = user_string($arg1)
14                 completequeries[query] <<< delta
15                 delete livequeries[t, $arg1]
16         }
17 }
```

Example 8 continued : pgtop.stp

Listing 10 – pgtop.stp (part 2 : printing)

```
19  probe timer.s(2) {
20          printf("%10s %10s %10s %10s %22s\n",
21                  "Count", "Total(us)", "Avg", "Max", "Query")
22          foreach (query in completequeries- limit 10) {
23                  printf("%10d %10d %10d %10d %22s\n",
24                          @count(completequeries[query]),
25                          @sum(completequeries[query]),
26                          @avg(completequeries[query]),
27                          @max(completequeries[query]),
28                          query)
29          }
30          delete completequeries
31  }
```

Example 8 continued : running pgtop.stp

```
# stap pgtop.stp
Count Total(us)     Avg     Max
    Query
6       1672      278     487 SELECT * FROM
    money_data;
5     270491    54098   90422 DELETE FROM
    money_data;
4       2759      689    1481 SELECT '' AS five, *
    FROM FLOAT8_TBL;
3      11074     3691   10034 SELECT * FROM
    enumtest WHERE col = 'orange';
3        696      232     410 SELECT '' AS five, *
    FROM FLOAT4_TBL;
2     122723    61361   66223 INSERT INTO
    VARCHAR_TBL (f1) VALUES ('a');
2     122176    61088  100516 INSERT INTO CHAR_TBL
    (f1) VALUES ('a');
2        273      136     144 SELECT max(col) FROM
    enumtest WHERE col < 'green';
2        313      156     192 SELECT max(col) FROM
    enumtest;
```

What is guru mode ?

- `stap -g`
- allows you to actually change things, not just observe
- set variables instead of just reading them
- embed custom C code about anywhere
- easy to mess up something and cause a crash

Example 9 : changing kernel state in Guru mode

Listing 11 – examples/general/keyhack.stp

```
1 # This is not useful, but it demonstrates
2 # that Systemtap can modify variables in a
3 # running kernel.
4 probe kernel.function("kbd_event") {
5     # Changes 'm' to 'b' .
6     if ($event_code == 50) $event_code = 48
7 }
```

Example 10 : another real support case

Customer Hello, my application mysteriously stops every 24 days. This seems to match the jiffies counter hitting 0x7fffffff. Can you help ?

Support Let me think about it.

- cannot write to /dev/mem above 1MB in RHEL x86 (see *[Crash-utility] Unable to change the content of memory using crash on a live system*)
- we could build a custom kernel
- or we could use `stap -g`

Example 10 continued : setting kernel variable

Listing 12 – set-jiffies.stp

```
1  %{
2          #include <linux/jiffies.h>
3  %}
4
5  function set_jiffies (value:long) %{
6          jiffies = THIS->value;
7  %}
8
9  probe begin {
10         set_jiffies($1)
11         exit()
12 }
```

It tends to hang the system for a few seconds and to lock up network drivers but it helped to reproduce and analyze the problem in minutes instead of weeks.

Example 11 : forbidding specific file names

Listing 13 – examples/general/badname.stp (simplified)

```
1  function filter:long (name:string) {
2      return euid() && isinstr(name, "XXX")
3  }
4
5  probe kernel.function("may_create@fs/namei.c").return
6  {
7      file_name = kernel_string($child->d_name->name)
8      if (!$return && filter(file_name))
9          $return = -13  # -EACCES (Permission denied)
10 }
```