Tracing and profiling production code with SystemTap

Adrien Kunysz
adk@redhat.com

Open Source Developers Conference, Paris, France
10 October 2010

Who is Adrien Kunysz?

- Senior Technical Support Engineer at Red Hat UK
  - when your sysadmin calls support, I pick up the phone
- co-founder of FSUGAr (Belgium)
- Krunch or adk on Freenode
- I like to look at core files and to read code
  - . . . but sometimes you need something more dynamic
- I am just a happy SystemTap user (not a developer)
- I am more into systems and low level stuff, not so much into web applications
  - . . . so let me know when I stop making sense

What are we going to discuss?

Explaining SystemTap

Practical Examples

Requirements and safety

Conclusion

What is SystemTap?

According to 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.*

- allows you to observe about anything about a running system
  - this includes your applications internals
- obviously designed for kernel and C people
  - . . . but recent developments makes it higher level friendly
- this talk doesn't cover kernel tracing at all

How does it work?

1. write or choose a script describing what you want to observe
2. stap translates it into a kernel module
3. stap loads the module and communicates with it
4. just wait for your data

The five stap passes

```
# 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.
[...script output goes here...]
Pass 5: run completed in 20usr/30sys/4204real ms.
```

SystemTap probe points examples

SystemTap is all about executing certain actions when hitting certain probe points.

- `process("/bin/ls").function("*")`
  - when entering any function in /bin/ls (not its libraries or syscalls)
- `process("/lib/libc.so.6").function("*malloc*")`
  - when entering any glibc function which has "malloc" in its name
- `timer.ms(200)`
  - every 200 milliseconds

More probe points examples

- `kernel.function("*init*")`, `kernel.function("*exit*").return`
  - when entering any kernel function which has "init" in its name or returning from any kernel function which has "exit" in its name
- `process("postgres").mark("query__start")`
  - when starting a PostgreSQL query
- `hotspot.method_entry`
  - when entering any Java method
- `python.function.return`
  - when returning from any Python function

See `man stapprobes` and `/usr/share/systemtap/tapset/` for more.

SystemTap programming language

- mostly C-style syntax with a feeling of awk
- builtin associative arrays
- builtin aggregates of statistical data
  - very easy to collect data and do statistics on it (average, min, max, count,...)
- many helper functions (builtin and in tapsets)

RTFM: *SystemTap Language Reference* shipped with SystemTap (langref.pdf)

Some helper functions examples

`pid()` which process is this?

`execname()` what is the name of this process?

`tid()` which thread is this?

`gettimeofday_s()` epoch time in seconds

`probefunc()` what function are we in?

`print_backtrace()` figure out how we ended up here (C only)

`print_jstack()` same for Java

There are many many more. RTFM (`man stapfuncs`) and explore `/usr/share/systemtap/tapset/`.

Some cool stap options

- `-x` trace only specified PID
- `-c` run given command and only trace it and its children
- `-L` list probe points matching given pattern along with available variables
- `-e` write your stap script on the command line
- `-g` embed C code in stap script
  - unsafe, dangerous and fun

Agenda

Explaining SystemTap

Practical Examples

Requirements and safety

Conclusion

Example 1: adding a printf in a C program

Listing 1: example1.c

```
1  #include <stdlib.h>
2
3  int plopify(int n) {  /* not sure we are reaching this? */
4          return n*n;
5  }
6
7  int main(int argc, char **argv) {
8          int n = argc > 1 ? atoi(argv[1]) : 0;
9          if (n % 2 == 0)
10                 n = plopify(n);
11         return n;
12 }
```

Listing 2: example1.stp

```
1  probe $1 {
2      printf("we hit the probe at %s\n",
           ctime(gettimeofday_s()))
3  }
```

Example 1 continued: adding a printf in a C program

```
1 $ cc -g -o example1 example1.c
2 $ stap -L 'process("example1").function("*")'
3 process("example1").function("main@example1.c:7") $argc:int
      $argv:char** $n:int
4 process("example1").function("plopify@example1.c:3") $n:int
5
6 # stap example1.stp
      'process("example1").function("plopify")' -c
      './example1 42'
7 we hit the probe at Sun Oct 10 11:19:40 2010
```

Example 1 continued: adding a printf in a C program

Want to add the `printf()` when exiting the function instead?

```
process("example1").function("plopify").return
```

When hitting a specific line in the file?

```
process("example1").statement("*@example1.c:10")
```

When hitting the third line within the `main()` function?

```
process("example1").statement("main@*+3")
```

When hitting every line between 9 and 11?

```
process("example1").statement("*@example1.c:9-11")
```

Example 2: 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 2 continued: watching Java threads

Listing 3: 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 3: 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 3 continued: call graph of a Python application

Listing 4: python-callgraph.stp

```
function trace(entry_p, funcname, filename, lineno) {
        if (isinstr(filename, @1)) {
                printf("%s%s%s@%s:%d\n",
                        thread_indent(entry_p),
                        (entry_p > 0 ? "->" : "<-" ),
                        funcname, filename, lineno)
        }
}

probe python.function.entry {
        trace(1, funcname, filename, lineno)
}

probe python.function.return {
        trace(-1, funcname, filename, lineno)
}
```

Example 3 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 4: 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 4 continued: pgtop.stp

Listing 5: 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 4 continued: pgtop.stp

Listing 6: 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 4 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;
```

Agenda

Requirements

To be able to trace application code, SystemTap relies on several things:

- you need Linux (the kernel)
- you need kprobes (`CONFIG_KPROBES=y`)
- you need the utrace kernel patch
  - not in mainline (yet?)
  - in Red Hat Enterprise Linux 5+, Fedora,...
  - not required if you are only interested in kernel probing
- you need debug symbols of the code you want to trace
  - `package-debuginfo` on RPM distros
  - `package-dbg` on .deb distros
  - build your C applications with `gcc -g`
  - not required if you like to read assembly
- for higher level languages, you need an instrumented runtime
  - what does this mean?

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
- language runtimes that are built and packaged properly in Fedora/RHEL6: Java (OpenJDK/IcedTea), Python, TCL
- Perl is almost there (`./Configure -Dusedtrace`)
- check build options and bug database of your favourite language

Performances and safety

- language-level safety features
  - no pointers
  - no unbounded loops
  - type inference
  - you can also write probe handlers in C (with -g) but don't complain if you break stuff
- runtime safety features
  - stap enforces maximum run time for each probe handler
  - various concurrency constraints are enforced
  - overload processing (don't allow stap to take up all the CPU time)
  - many things can be overriden manually if you really want
  - see `SAFETY AND SECURITY` section of `stap(1)`

The overhead depends a lot of what you are trying to do but in general stap will try to stop you from doing something stupid (but then you can still force it to do it).

References and questions

- SystemTap wiki: http://sourceware.org/systemtap/wiki
- lot of excellent documentation included (for kernel probing at least):
  - `man -k stap`
  - file:///usr/share/doc/systemtap*
- lot of examples for system tracing and debugging: http://sourceware.org/systemtap/examples/
- userland/application tracing examples and video demos:
  - sourceware.org/systemtap/wiki/UsingStaticUserMarkers
  - sourceware.org/systemtap/wiki/JavaMarkers
  - fedoraproject.org/wiki/Features/SystemtapStaticProbes
- systemtap@sources.redhat.com
- irc://chat.freenode.net/#systemtap