FOUR USEFUL DTRACE SCRIPTS YOU CAN PORT TO SYSTEMTAP

DOMINIC DUVAL, SENIOR CONSULTANT, RED HAT CONSULTING.

TABLE OF CONTENTS

INTRODUCTION

A persistent myth among system administrators is that there is no DTrace equivalent for Red Hat® Enterprise Linux®. While this was true a few years ago, today, SystemTap can accomplish the same tasks as DTrace. This paper describes in detail how to accomplish the same tasks with SystemTap that some of the most popular DTrace scripts perform. We found four DTrace scripts that are useful in several situations and adapted some existing SystemTap scripts to achieve similar results. By using SystemTap to react to common scenarios that system administrators and developers face, this guide shows how powerful SystemTap can be.

1. TOP I/O SCRIPT

Original DTrace script: www.brendangregg.com/DTrace/iotop

When investigating performance issues, being able to monitor how applications use storage devices can be critical. Knowing which programs transfer the largest amount of data is key. Out of dozens of processes, we want to identify the most active ones, with a screen that looks as follows:

```
     process      read   KB tot      write   KB tot

  userscript    488692   244346     488692   244346

        sshd     11912     1220        303     1234

        find         8        4      17799     1201

          vi       206      734         12        5

   automount         4        1          0        0

        bash        40        0         98        1

      stapio       151        0          2        0

   mcstransd         1        0          0        0
```

From the screen above it should be clear that `userscript` is the first process to investigate if we're running into some performance issues with our storage devices. Applying sensors at the right place with SystemTap will help us get this data easily. In this case, what we need to do is simple: increase counters whenever data is read or written.

We first need to setup three arrays: one for the amount of data read, another for the amount of data written, and a last one for the total amount of data transfered per process. We will use the process name as an index in those arrays to present data for each process later on.

GLOBAL READS, WRITES, TOTALS

Now we insert some probes in the kernel. Linux relies on system calls to let applications communicate with the kernel and the hardware. System calls are numbered and named. This is in fact the first challenge associated with SystemTap, just like DTrace: we need to find the name of the system call we're interested in. Since the whole point behind this script is to monitor data transfered with storage devices, we'll focus on the `read()` and `write()` system calls.

To make things easier, SystemTap provides a number of existing hooks that we can use in our script. These are called tapsets. In our case, we're interested in the `syscall` tapset. More specifically, in the `read` and `write` hooks. We choose to catch those two system calls when they return, hence the `.return` at the end of the call:

```
probe syscall.read.return {
  count = $return
  if ( count >= 0 ) {
    e=execname();
    reads[e] <<< count # statistics array
    totals[e] += count
  }
}

probe syscall.write.return {
  count = $return
  if (count >= 0 ) {
    e=execname();
    writes[e] <<< count # statistics array
    totals[e] += count
```

Some variables are already defined for us here; notice that `$return` never got declared anywhere. The `syscall` tapset defined `$return` automatically, so we can use it to access the return value of the system call. In the case of `read` and `write`, the return value is the number of bytes that got pulled from or pushed to storage. Exactly what we need.

`execname()` is part of another tapset. It returns the name of the process on behalf of which the system call is executed. Things like 'ls,' 'httpd,' or anything that can be executed on your system. This might sound strange to programmers, but we will use this name as an index for the arrays we built previously. It makes our job a lot easier.

Also note the use of the "<<<" operator instead of "=". We call this statistics accumulation. This feature enables us to compute the sum, average, min, and max values quickly.

A SystemTap script can run for a while. How long? As long as you want. SystemTap listens to the `SIGINT` signal. (This is CTRL-C for people who like to use keyboards.) The script automatically stops when you press those keys. In fact, it does more than that; it also executes the `probe end` statement if you provided one. As a matter of fact, we did:

```
  probe end {
   printf("%16s %8s %8s %8s %8s\n",
     "process", "read", "KB tot", "write", "KB tot")
   foreach (name in totals- limit 20) { # sort by total io
     printf("%16s %8d %8d %8d %8d\n",
       name,
       @count(reads[name]),
       (@count(reads[name]) ? @sum(reads[name])>>10 : 0 ),
       @count(writes[name]),
       (@count(writes[name]) ? @sum(writes[name])>>10 : 0 )
     )
   }
 }
```

Pay attention to the `foreach` loop above. This construct will let us extract the name of the 20 most active processes, based on the total number of bytes transfered. That's what we've been recording earlier in the totals array.

The `@count` and `@sum` are data extraction functions. We use them to analyze statistics we built with the "<<<" operator. We want to print the number of `read` and `write` operations, as well as the total number of bytes read and written for those top 20 processes.

The following construct might give you a hard time:

```
(@count(reads[name]) ? @sum(reads[name])>>10 : 0 )
```

It's really just a compact way to print the sum of all values stored in the reads array, as long as there's actually something in there. Otherwise we'll just print 0.

Once the entire script described above gets saved in a file--we usually name it something like `iotop.stp`--we're ready to execute it:

```
# stap iotop.stp
```

Parsing and compiling the script will take one or two seconds. Then it will start executing. Remember to press CTRL-C to see results on the screen.

2. SYSTEM WIDE ERRORS SCRIPT

Original DTrace script: www.brendangregg.com/DTrace/errinfo

Applications need to handle errors all the time. That's generally fine; however, large numbers of errors may lead to other issues. They can drop performance. Or indicate that something is wrong with your system. Perhaps point at a bug in your software.

DTrace provides a clever script that produces an overview of all errors encountered on the system through system calls. Turns out, we can do the same with SystemTap. Errors are generated in the kernel from those functions called system calls. There are a few hundreds of those. An easy way to catch all of them is to probe `syscall.*` Adding `.return` will ensure that the probe always gets executed when a system call returns, rather than when it starts:

```
global execname, errors
probe syscall.*.return {
  errno = $return
  thissyscall=probefunc();
  if ( errno < 0 ) {
    p = pid()
    execname[p]=execname();
    errors[p, errno, thissyscall] ++
  }
}
```

Catching the return value of a function (any function, really) with SystemTap is trivial--just refer to `$return`. That variable is built in; you don't need to declare it. Also notice the calls to the `probefunc()` and `execname()`. `probefunc()` returns the name of the function we're probing; we used a * in the probe name, so we need to figure this out. execname() gives us the name of the executable that invoked this system call. All that information will be displayed later on.

The errors array is the core of this script. We're now working with three dimensional arrays. That's what we'll use to record the number of hits for any error number returned by a specific system call for any given process on the system.

As always, whenever we hit CTRL-C, the end probe gets invoked:

```
probe end {
  printf(“%8s %16s %16s %12s %8s\n”,
    “PID”, “Syscall”, “Process”, “Error”, “Count”)
  foreach ([pid, error, thissyscall] in errors- limit 20) {
    printf(“%8d %16s %16s %12s %8d\n”,
      pid,
      thissyscall,
      execname[pid],
      error ? errno_str(error) : “”,
      errors[pid, error, thissyscall]
    )
  }
```

There's one new aspect to consider here; printing error numbers is good, but error messages are even better. SystemTap provides an array called `errno_str` that contains actual error messages, making the whole thing easier to understand. Just use the error number as the index, and you'll get the message as a result:

```
# stap errno.stp
     PID        Syscall      Process     Error       Count
   19793      sys_close      bash        EBADF       86
    3225      sys_futex      automount   ETIMEDOUT   52
   19793    sys_newstat      bash        ENOENT      21
   19793      sys_wait4      bash        ECHILD      12
    3510    sys_newstat      hald        ENOENT      8
   19791    sys_connect      sshd        ENOENT      6
   19795       sys_open      egrep       ENOENT      6
   19797       sys_open      egrep       ENOENT      6
   19799       sys_open      egrep       ENOENT      6
```

3. BLOCK DEVICE SEEKS SCRIPT

Back to storage devices. These tend to be the devices that slow down entire systems when not used appropriately. One reason why a storage device, such as a disk, might be slow could be seek usage. In other words, applications force the disk to fetch data on vastly different locations on the disk. This can be hard to tell; statistics on disk seeks are not easily accessible from standard interfaces. Once again, SystemTap makes it trivial.

The plan here is to monitor every data access at the disk level. We'll do this by attaching a probe to the request function. Every storage device driver has such a function. It gets executed whenever a read or write operation needs to be serviced by the hardware device. First, we need to set up two variables: `seeks`, which contains data for all seek operations on a specific device, and `oldsec`, which holds the last sector accessed on a device.

GLOBAL SEEKS, OLDSEC

The guts of the script lies in the `ioblock.request` probe. This probe will automatically let you monitor all request functions on the system. Incidentally, this is part of the `ioblock` tapset, and you can refer to all tapsets by looking at the `/usr/share/systemtap/tapset` directory. Everything in this directory can be used in your scripts.

Whenever we execute the request function, we will read the `sector` variable. This contains the sector number where the operation starts. By computing the difference between the current sector and one accessed in the last request invocation, we'll be able to compute the seeks value for this device. This can be a positive or negative number. We'll also save the current sector number for next time we execute a request.

```
probe ioblock.request {

  sec = sector
  seeks[devname] <<< sec - oldsec[devname]
  oldsec[devname] = sector

}
```

Finally, when the script ends, we need to print a summary of what happened. We chose to take the top five most active devices on the system. For each of them we'll print a histogram: `@hist_log` will automatically take the data accumulated in seeks and summarize it on the screen as a histogram based on a logarithmic scale (just remove `_log` to switch to a linear scale).

```
probe end {

  foreach ([devname] in seeks- limit 5) {
    printf("Device: %s\n", devname)
    println(@hist_log(seeks[devname]))
  }
}
```

4. TCPTOP SCRIPT

Original DTrace script: www.brendangregg.com/DTrace/tcptop

Wouldn't it be nice if something could tell us what application network traffic comes from? SystemTap can.

This script monitors the TCP/IP stack and looks for packets received from and sent to the application layer (i.e. sockets). That way we can see easily which process, user, and command transfered data on the network, how much data was transfered, and in what direction it went.

Other than process IDs, command names, and user IDs, we'll need to keep track of two critical pieces of data: bytes transmitted and received for every process we probed:

```
global xmit, recv, process, execname, user
```

We want to probe the socket probe points. Two probes are of interest here: `socket.send` and `socket.receive`. These are documented in the `stapprobes.socket` main page, as well as other probe points related to networking.probe socket.send

```
{
        p = pid()
        execname[p] = execname()
        user[p] = uid()
        xmit[p] <<< size
        process[p] ++
}

probe socket.receive
{
        p = pid()
        execname[p] = execname()
        user[p] = uid()
        recv[p] <<< size
        process[p] ++
```

The two probes are nearly identical: we want to keep track of how many bytes are transmitted and received for every process that deals with the network. As seen previously, the `pid()`, `execname()`, and `uid()` functions let us easily retrieve the process ID, the command name, and the user ID associated with the probed function.

One major difference from the scripts we've covered so far lies in the way we print data. Here we make use of a timer. This will let us invoke the script and monitor results while it keeps running. The function that prints the data is relatively similar to what we've used so far:

```
function print_activity()
{
        printf("%5s %5s %7s %7s %7s %7s %-15s\n",
                "PID", "UID", "XMIT_PK", "RECV_PK",
                "XMIT_KB", "RECV_KB", "COMMAND")

        foreach ([pid] in process-) {
                n_xmit = @count(xmit[pid])
                n_recv = @count(recv[pid])
                printf("%5d %5d %7d %7d %7d %7d %-15s\n",
                        pid, user[pid], n_xmit, n_recv,
                        n_xmit ? @sum(xmit[pid])/1024 : 0,
                        n_recv ? @sum(recv[pid])/1024 : 0,
                        execname[pid])
        }
```

```
print("\n")

        delete execname
        delete user
        delete xmit
        delete recv
        delete process
```

Notice the `delete` statements; these let us reset all counters we use in this script so that current results are not affected by the last sample. In this case, we're taking samples every five seconds by attaching the `print_activity()` function to the timer probe:

```
probe timer.ms(5000)
{
        print_activity()
)
```

Final result (assuming there's some kind of network traffic on your system) should look as follows, repeated every five seconds:

```
PID   UID XMIT_PK RECV_PK XMIT_KB RECV_KB COMMAND
11404    0       0    3658       0    4879 wget
11004   48       1       2       4       0 httpd
11005   48       1       2       0       0 httpd
10270    0       1       0       0       0 sshd
```

CONCLUSION

Just a handful of SystemTap scripts were covered in this document. Nonetheless, these should give you a good idea what the typical structure of a script looks like. With just a few lines of code, system administrators and developers can implement scripts that are just as capable as their DTrace equivalents. Several topics, such as embedded C code, kernel markers, as well as advanced features of the SystemTap language were not discussed. We encourage you to visit the SystemTap web site, which is always a great source of information and sample scripts: www.sourceware.org/systemtap/.

LEARN MORE ABOUT RED HAT CONSULTING

www.redhat.com/consulting

EUROPE, MIDDLE EAST AND AFRICA
00800 7334 2835
www.europe.redhat.com
europe@redhat.com

Turkey: 00800 448 820 640
Israel: 1809 449 548
UAE: 80004449549

ASIA PACIFIC
+65 6490 4200
www.apac.redhat.com
apac@redhat.com

ASEAN: 800 448 1430
Australia and New Zealand: 1800 733 428
Greater China: 800 810 2100
India: +91 22 3987 8888
Japan: 0120 266 086
Korea: 080 708 0880

NORTH AMERICA
1-888-REDHAT1
www.redhat.com

LATIN AMERICA
+54 11 4341 6200
www.latam.redhat.com
info-latam@redhat.com

www.redhat.com
#1098170_0609