SystemTap
“painful to use”, but more painful not to
Logo credit: Andy Fitzsimon

Logo ref: http://sourceware.org/systemtap/wiki/LW2008SystemTapTutorial

# SystemTap Tutorial - Part 1

Who is doing maximum read/write on my server?
Can I add some debug statements in the kernel without rebuilding, rebooting the system?

You might have asked these question to yourself, if you are a System Administrator or a Kernel Developer.

Lets see what are our choices to answer above questions:-

**Tracing** - Provides info while running and gives quick overview of code flow but gives lot of information. Tools like **strace, ltrace** and **ftrace** are used for tracing.

**Profiling** - It does the sampling while running and we can do the analysis after the event has occurred. **Oprofile** is used for sampling.

**Debugging** - We can set breakpoints, look at the variables, memory, registers stack trace etc.We can debug only one program at a time and debugger stops it while we do the inspection. **GDB/KDB** is used for such debugging.

So, which of the above mentioned tool you will use. You might be thinking of using combination of above mentioned tools. Won't it be great to have all the capabilities from above tools in one tool?

**Welcome to SystemTap!!**
SystemTap can monitor system wide multiple synchronous and asynchronous events at the same time. It can do scriptable filtering and statistics collection. Its a dynamic method of monitoring and tracing the operations of a running Linux kernel.

To instrument the running kernel SystemTap uses ***Kprobes*** and ***return probes***.
With kernel debug information it gets the addresses for functions and variables referenced in the script. With utrace systemtap supports probing user-space executables and shared libraries as well.

SystemTap is useful to System Administrators, Kernel Developers, Support Engineers, Researchers and students.

## Installation

On Fedora

```
yum install systemtap kernel-devel
yum --enablerepo=fedora-debuginfo --enablerepo=updates-debuginfo install
```

Similarly to use SystemTap on Ubuntu or any other distro you need to install the systemtap package and corresponding kernel's debuginfo packages.

You need to be a root user to run the SystemTap scripts. You can also add a normal user to either "**stapdev**" or "**stapusr**" group to allow him/her to run the script.

## How it works?

To understand it let run a script in verbose mode (-v).

```
$ stap -v -e 'probe syscall.read {printf("syscall %s  arguments %s \n", name, argstr);
exit()}'
Pass 1: parsed user script and 65 library script(s) using 83596virt/20428res/2412shr kb,
in 150usr/10sys/249real ms.
Pass 2: analyzed script: 1 probe(s), 4 function(s), 0 embed(s), 0 global(s) using
216260virt/115660res/73964shr kb, in 560usr/20sys/946real ms.
Pass 3: translated to C into
"/tmp/stapUGVeZi/stap_b40c8268c87acc683f75ded62a52ee66_2113.c" using
216260virt/117180res/75484shr kb, in 320usr/40sys/1014real ms.
Pass 4: compiled C into "stap_b40c8268c87acc683f75ded62a52ee66_2113.ko" in
3010usr/1210sys/12818real ms.
Pass 5: starting run.
syscall read arguments 4, 0x00007fffa773b4c0, 8196
Pass 5: run completed in 20usr/60sys/174real ms.
```

The **stap** program is the **front-end** to the SystemTap tool. '**-e**' tells it to run a script from the next argument.

Pass 1-2
    Parse the script and the code is checked for semantic and syntax errors.
    Any tapset reference is imported. Debug data provided via debuginfo packages are read to find addresses for functions and variables referenced in the script.
Pass 3
    Translate the script into C code.
Pass 4
    Compile the translated C code and create a kernel module.
Pass 5
    Insert the module in the kernel.

Once the module is loaded, probes are inserted at proper locations. From now on whenever a probe is hit, handler for that probe is called.

Syntax to write an event and its handler.

**probe <event> { handler }**

Where **event** is kernel.function, process.statement, timer.ms, begin, end, (tapset) aliases. For more info look at the man page of "**stapprobes"**.

**handler** can have:
- filtering/conditionals (if ... next)
- control structures (foreach, while)

In the script you don't need to declare the type of variable. It is inferred from the context. To make our life easier helper functions like pid, execname, log etc are defined. Look at the language reference guide for more info. If you have installed the package then you can find it at "/usr/share/doc/systemtap-<*version*>/langref.pdf".

**How to Run**

```
stap -e '<script>' [-c <target program>]
stap script.stp [-c <target program>]
stap -l '<event*>'
```

**Tapset Libraries**

In the example shown earlier after probing on read system call we printed the name of the system call and the arguments passed to via "name" and "argstr".

This was possible because in the one of the tapset library "/usr/share/systemtap/tapset/syscalls2.stp", following is defined.

```
probe syscall.read = kernel.function("SyS_read").call !,
                kernel.function("sys_read").call
{
      name = "read"
      fd = $fd
```

```
        buf_uaddr = $buf
        count = $count
        argstr = sprintf("%d, %p, %d", $fd, $buf, $count)
}
```

Tapsets provide abstraction to common probe points and define functions, which you can use in your script. They are not runnable (probe aliases not probes).

**Examples**

```
$ cat syscount.stp
global syscalls
probe syscall.* { syscalls[name] += 1 }
probe timer.s(10) {
        foreach(n in syscalls- limit 5)
        printf("%s = %d\n", n, syscalls[n])
        delete syscalls
}
```

Here we have taken an associative array *syscalls.* **Associative array** is a collection of unique keys; each key in the array has a value associated with it.  Here system call name would be an unique key.

Whenever a system call  is called, we increment the value of corresponding value in the array.Then after 10 sec we print the top 5 system calls, which were called.

```
 $ stap syscount.stp
read = 116
poll = 55
ppoll = 49
setitimer = 24
writev = 22
```

Let's look at another script from which we want get the process name and pid of the process who calls the maximum system calls. We also don't want to take SystemTap process which launches the script(stapio),  into consideration.

```
$ cat syscount_per_process.stp
global syscalls
probe syscall.* {
        if (execname() == "stapio")
                next
        syscalls[execname(), pid()] += 1
}
probe timer.s(10) {
        foreach([procname, id] in syscalls- limit 5)
        printf("%s[%d] = %d\n", procname, id, syscalls[procname, id])
        delete syscalls
}
```

To immediately return from the enclosing probe handled we use *next* statement.

```
$ stap syscount_per_process.stp
hald-addon-stor[1074] = 30
sendmail[1157] = 14
rtkit-daemon[1387] = 8
gdm-simple-gree[1374] = 8
gnome-power-man[1370] = 7
```

We can do other interesting stuff like aggregation, getting a call graph,  modifying a kernel variable in the running kernel etc. This all we cover in next month.

**References**

http://sourceware.org/systemtap/
http://sourceware.org/systemtap/wiki/LW2008SystemTapTutorial
http://sourceware.org/systemtap/wiki/HomePage?action=AttachFile&do=view&target=fosdem-stap.pdf
http://www.redbooks.ibm.com/redpapers/pdfs/redp4469.pdf