Systemtap tutorial

Frank Ch. Eigler <fche@redhat.com>
October 28, 2016

Contents

1 Introduction **2**

2 Tracing **2**

2.1 Where to probe . . . 3
2.2 What to print . . . 4
2.3 Exercises . . . 5

3 Analysis **6**

3.1 Basic constructs . . . 6
3.2 Target variables . . . 6
3.3 Functions . . . 7
3.4 Arrays . . . 9
3.5 Aggregates . . . 9
3.6 Safety . . . 10
3.7 Exercises . . . 11

4 Tapsets **11**

4.1 Automatic selection . . . 11
4.2 Probe point aliases . . . 12
4.3 Embedded C . . . 14
4.4 Naming conventions . . . 14
4.5 Exercises . . . 15

5 Further information **16**

A Glossary **16**

B Errors **16**

B.1 Parse errors . . . 16
B.2 Type errors . . . 16
B.3 Symbol errors . . . 17
B.4 Probing errors . . . 17

B.5 Runtime errors 18

C Acknowledgments **18**

1 Introduction

Systemtap is a tool that allows developers and administrators to write and reuse simple scripts to deeply examine the activities of a live Linux system. Data may be extracted, filtered, and summarized quickly and safely, to enable diagnoses of complex performance or functional problems.

NOTE: This tutorial does not describe every feature available in systemtap. Please see the individual `stap` manual pages for the most up-to-date information. These may be available installed on your system, or at `http://sourceware.org/systemtap/man/`.

The essential idea behind a systemtap script is to name *events*, and to give them *handlers*. Whenever a specified event occurs, the Linux kernel runs the handler as if it were a quick subroutine, then resumes. There are several kind of events, such as entering or exiting a function, a timer expiring, or the entire systemtap session starting or stopping. A handler is a series of script language statements that specify the work to be done whenever the event occurs. This work normally includes extracting data from the event context, storing them into internal variables, or printing results.

Systemtap works by translating the script to C, running the system C compiler to create a kernel module from that. When the module is loaded, it activates all the probed events by hooking into the kernel. Then, as events occur on any processor, the compiled handlers run. Eventually, the session stops, the hooks are disconnected, and the module removed. This entire process is driven from a single command-line program, `stap`.

```
# cat hello-world.stp
probe begin
{
  print ("hello world\n")
  exit ()
}

# stap hello-world.stp
hello world
```

Figure 1: A systemtap smoke test.

This paper assumes that you have installed systemtap and its prerequisite kernel development tools and debugging data, so that you can run the scripts such as the simple one in Figure 1. Log on as `root`, or even better, login as a user that is a member of `stapdev` group or as a user authorized to `sudo`, before running systemtap.

2 Tracing

The simplest kind of probe is simply to *trace* an event. This is the effect of inserting strategically located `print` statements into a program. This is often the first step of problem solving: explore by seeing a history of what has happened.

```
# cat strace-open.stp
probe syscall.open
{
  printf ("%s(%d) open (%s)\n", execname(), pid(), argstr)
}
probe timer.ms(4000) # after 4 seconds
{
  exit ()
}

# stap strace-open.stp
vmware-guestd(2206) open ("/etc/redhat-release", O_RDONLY)
hald(2360) open ("/dev/hdc", O_RDONLY|O_EXCL|O_NONBLOCK)
hald(2360) open ("/dev/hdc", O_RDONLY|O_EXCL|O_NONBLOCK)
hald(2360) open ("/dev/hdc", O_RDONLY|O_EXCL|O_NONBLOCK)
df(3433) open ("/etc/ld.so.cache", O_RDONLY)
df(3433) open ("/lib/tls/libc.so.6", O_RDONLY)
df(3433) open ("/etc/mtab", O_RDONLY)
hald(2360) open ("/dev/hdc", O_RDONLY|O_EXCL|O_NONBLOCK)
```

Figure 2: A taste of systemtap: a system-wide `strace`, just for the `open` system call.

This style of instrumentation is the simplest. It just asks systemtap to print something at each event. To express this in the script language, you need to say where to probe and what to print there.

2.1 Where to probe

Systemtap supports a number of built-in events. The library of scripts that comes with systemtap, each called a "tapset", may define additional ones defined in terms of the built-in family. **See the `stapprobes` man page for details on these and many other probe point families.** All these events are named using a unified syntax with dot-separated parameterized identifiers:

`begin`	The startup of the systemtap session.
`end`	The end of the systemtap session.
`kernel.function("sys_open")`	The entry to the function named `sys_open` in the kernel.
`syscall.close.return`	The return from the `close` system call.
`module("ext3").statement(0xdeadbeef)`	The addressed instruction in the `ext3` filesystem driver.
`timer.ms(200)`	A timer that fires every 200 milliseconds.
`timer.profile`	A timer that fires periodically on every CPU.
`perf.hw.cache_misses`	A particular number of CPU cache misses have occurred.
`procfs("status").read`	A process trying to read a synthetic file.
`process("a.out").statement("*@main.c:200")`	Line 200 of the `a.out` program.

Let's say that you would like to trace all function entries and exits in a source file, say `net/socket.c` in the kernel. The `kernel.function` probe point lets you express that easily, since systemtap examines the kernel's debugging information to relate object code to source code. It works like a debugger: if you can name or place it, you can probe it. Use `kernel.function("*@net/socket.c").call` for the function entries[1], and `kernel.function("*@net/socket.c").return` for matching exits. Note the use of wildcards in the function name part, and the subsequent `@FILENAME` part. You can also put wildcards into the file name, and

[1] Without the `.call` qualifier, inlined function instances are also probed, but they have no corresponding `.return`.

even add a colon (`:`) and a line number, if you want to restrict the search that precisely. Since systemtap will put a separate probe in every place that matches a probe point, a few wildcards can expand to hundreds or thousands of probes, so be careful what you ask for.

Once you identify the probe points, the skeleton of the systemtap script appears. The `probe` keyword introduces a probe point, or a comma-separated list of them. The following `{` and `}` braces enclose the handler for all listed probe points.

```
probe kernel.function("*@net/socket.c") { }
probe kernel.function("*@net/socket.c").return { }
```

You can run this script as is, though with empty handlers there will be no output. Put the two lines into a new file. Run `stap -v FILE`. Terminate it any time with `^C`. (The `-v` option tells systemtap to print more verbose messages during its processing. Try the `-h` option to see more options.)

2.2 What to print

Since you are interested in each function that was entered and exited, a line should be printed for each, containing the function name. In order to make that list easy to read, systemtap should indent the lines so that functions called by other traced functions are nested deeper. To tell each single process apart from any others that may be running concurrently, systemtap should also print the process ID in the line.

Systemtap provides a variety of such contextual data, ready for formatting. They usually appear as function calls within the handler, like you already saw in Figure 1. **See the `function::*` man pages for those functions and more defined in the tapset library**, but here's a sampling:

`tid()`	The id of the current thread.
`pid()`	The process (task group) id of the current thread.
`uid()`	The id of the current user.
`execname()`	The name of the current process.
`cpu()`	The current cpu number.
`gettimeofday_s()`	Number of seconds since epoch.
`get_cycles()`	Snapshot of hardware cycle counter.
`pp()`	A string describing the probe point being currently handled.
`ppfunc()`	If known, the the function name in which this probe was placed.
`$$vars`	If available, a pretty-printed listing of all local variables in scope.
`print_backtrace()`	If possible, print a kernel backtrace.
`print_ubacktrace()`	If possible, print a user-space backtrace.

The values returned may be strings or numbers. The `print()` built-in function accepts either as its sole argument. Or, you can use the C-style `printf()` built-in, whose formatting argument may include `%s` for a string, `%d` for a number. `printf` and other functions take comma-separated arguments. Don't forget a `"\n"` at the end. There exist more printing / formatting functions too.

A particularly handy function in the tapset library is `thread_indent`. Given an indentation delta parameter, it stores internally an indentation counter for each thread (`tid()`), and returns a string with some generic trace data plus an appropriate number of indentation spaces. That generic data includes a timestamp (number of microseconds since the initial indentation for the thread), a process name and the thread id itself. It therefore gives an idea not only about what functions were called, but who called them, and how long they took. Figure 3 shows the finished script. It lacks a call to the `exit()` function, so you need to interrupt it with `^C` when you want the tracing to stop.

```
# cat socket-trace.stp
probe kernel.function("*@net/socket.c").call {
  printf ("%s -> %s\n", thread_indent(1), ppfunc())
}
probe kernel.function("*@net/socket.c").return {
  printf ("%s <- %s\n", thread_indent(-1), ppfunc())
}

# stap socket-trace.stp
     0 hald(2632): -> sock_poll
    28 hald(2632): <- sock_poll
[...]
     0 ftp(7223): -> sys_socketcall
  1159 ftp(7223):  -> sys_socket
  2173 ftp(7223):   -> __sock_create
  2286 ftp(7223):    -> sock_alloc_inode
  2737 ftp(7223):    <- sock_alloc_inode
  3349 ftp(7223):    -> sock_alloc
  3389 ftp(7223):    <- sock_alloc
  3417 ftp(7223):   <- __sock_create
  4117 ftp(7223):   -> sock_create
  4160 ftp(7223):   <- sock_create
  4301 ftp(7223):   -> sock_map_fd
  4644 ftp(7223):    -> sock_map_file
  4699 ftp(7223):    <- sock_map_file
  4715 ftp(7223):   <- sock_map_fd
  4732 ftp(7223):  <- sys_socket
  4775 ftp(7223): <- sys_socketcall
[...]
```

Figure 3: Tracing and timing functions in `net/sockets.c`.

2.3 Exercises

1. Use the `-L` option to systemtap to list all the kernel functions named with the word "nit" in them.

2. Trace some system calls (use `syscall.NAME` and `.return` probe points), with the same `thread_indent` probe handler as in Figure 3. Print parameters using `$$parms` and `$$return`. Interpret the results.

3. Change figure 3 by removing the `.call` modifier from the first probe. Note how function entry and function return now don't match anymore. This is because now the first probe will match both normal function entry and inlined functions. Try putting the `.call` modifier back and add another probe just for `probe kernel.function("*@net/socket.c").inline` What `printf` statement can you come up with in the probe handler to show the inlined function entries nicely in between the `.call` and `.return` thread indented output?

3 Analysis

Pages of generic tracing text may give you enough information for exploring a system. With systemtap, it is possible to analyze that data, to filter, aggregate, transform, and summarize it. Different probes can work together to share data. Probe handlers can use a rich set of control constructs to describe algorithms, with a syntax taken roughly from `awk`. With these tools, systemtap scripts can focus on a specific question and provide a compact response: no `grep` needed.

3.1 Basic constructs

Most systemtap scripts include conditionals, to limit tracing or other logic to those processes or users or *whatever* of interest. The syntax is simple:

`if (`*EXPR*`)` *STATEMENT* `[else` *STATEMENT*`]`	if/else statement
`while (`*EXPR*`)` *STATEMENT*	while loop
`for (`*A*`;` *B*`;` *C*`)` *STATEMENT*	for loop

Scripts may use `break/continue` as in C. Probe handlers can return early using `next` as in `awk`. Blocks of statements are enclosed in `{` and `}`. In systemtap, the semicolon (`;`) is accepted as a null statement rather than as a statement terminator, so is only rarely[2] necessary. Shell-style (`#`), C-style (`/* */`), and C++-style (`//`) comments are all accepted.

Expressions look like C or `awk`, and support the usual operators, precedences, and numeric literals. Strings are treated as atomic values rather than arrays of characters. String concatenation is done with the dot (`"a" . "b"`). Some examples:

`(uid() > 100)`	probably an ordinary user
`(execname() == "sed")`	current process is sed
`(cpu() == 0 && gettimeofday_s() > 1140498000)`	after Feb. 21, 2006, on CPU 0
`"hello" . " " . "world"`	a string in three easy pieces

Variables may be used as well. Just pick a name, assign to it, and use it in expressions. They are automatically initialized and declared. The type of each identifier – string vs. number – is automatically inferred by systemtap from the kinds of operators and literals used on it. Any inconsistencies will be reported as errors. Conversion between string and number types is done through explicit function calls.

`foo = gettimeofday_s()`	foo is a number
`bar = "/usr/bin/" . execname()`	bar is a string
`c++`	c is a number
`s = sprint(2345)`	s becomes the string ”2345”

By default, variables are local to the probe they are used in. That is, they are initialized, used, and disposed of at each probe handler invocation. To share variables between probes, declare them global anywhere in the script. Because of possible concurrency (multiple probe handlers running on different CPUs), each global variable used by a probe is automatically read- or write-locked while the handler is running.

3.2 Target variables

A class of special “target variables” allow access to the probe point context. In a symbolic debugger, when you’re stopped at a breakpoint, you can print values from the program’s context. In systemtap scripts, for

[2] Use them between consecutive expressions that place unary +,− or mixed pre/post ++,−− in an ambiguous manner.

```
# cat timer-jiffies.stp
global count_jiffies, count_ms
probe timer.jiffies(100) { count_jiffies ++ }
probe timer.ms(100) { count_ms ++ }
probe timer.ms(12345)
{
  hz=(1000*count_jiffies) / count_ms
  printf ("jiffies:ms ratio %d:%d => CONFIG_HZ=%d\n",
    count_jiffies, count_ms, hz)
  exit ()
}

# stap timer-jiffies.stp
jiffies:ms ratio 30:123 => CONFIG_HZ=243
```

Figure 4: Experimentally measuring CONFIG_HZ.

those probe points that match with specific executable point (rather than an asynchronous event like a timer), you can do the same.

In addition, you can take their address (the & operator), pretty-print structures (the $ and $$ suffix), pretty-print multiple variables in scope (the $$vars and related variables), or cast pointers to their types (the @cast operator), or test their existence / resolvability (the @defined operator). Read about these in the manual pages.

To know which variables are likely to be available, you will need to be familiar with the kernel source you are probing. In addition, you will need to check that the compiler has not optimized those values into unreachable nonexistence. You can use `stap -L PROBEPOINT` to enumerate the variables available there.

Let's say that you are trying to trace filesystem reads/writes to a particular device/inode. From your knowledge of the kernel, you know that two functions of interest could be `vfs_read` and `vfs_write`. Each takes a `struct file *` argument, inside there is either a `struct dentry *` or `struct path *` which has a `struct dentry *`. The `struct dentry *` contains a `struct inode *`, and so on. Systemtap allows limited dereferencing of such pointer chains. Two functions, `user_string` and `kernel_string`, can copy `char *` target variables into systemtap strings. Figure 5 demonstrates one way to monitor a particular file (identified by device number and inode number). The script selects the appropriate variants of `dev_nr` and`inode_nr` based on the kernel version. This example also demonstrates passing numeric command-line arguments (`$1` etc.) into scripts.

3.3 Functions

Functions are conveniently packaged reusable software: it would be a shame to have to duplicate a complex condition expression or logging directive in every placed it's used. So, systemtap lets you define functions of your own. Like global variables, systemtap functions may be defined anywhere in the script. They may take any number of string or numeric arguments (by value), and may return a single string or number. The parameter types are inferred as for ordinary variables, and must be consistent throughout the program. Local and global script variables are available, but target variables are *not*. That's because there is no specific debugging-level context associated with a function.

A function is defined with the keyword `function` followed by a name. Then comes a comma-separated formal argument list (just a list of variable names). The `{ }`-enclosed body consists of any list of statements,

```
# cat inode-watch.stp
probe kernel.function ("vfs_write"),
      kernel.function ("vfs_read")
{
  if (@defined($file->f_path->dentry)) {
    dev_nr = $file->f_path->dentry->d_inode->i_sb->s_dev
    inode_nr = $file->f_path->dentry->d_inode->i_ino
  } else {
    dev_nr = $file->f_dentry->d_inode->i_sb->s_dev
    inode_nr = $file->f_dentry->d_inode->i_ino
  }

  if (dev_nr == ($1 << 20 | $2) # major/minor device
      && inode_nr == $3)
    printf ("%s(%d) %s 0x%x/%u\n",
      execname(), pid(), ppfunc(), dev_nr, inode_nr)
}
# stat -c "%D %i" /etc/crontab
fd03 133099
# stap inode-watch.stp 0xfd 3 133099
more(30789) vfs_read 0xfd00003/133099
more(30789) vfs_read 0xfd00003/133099
```

Figure 5: Watching for reads/writes to a particular file.

including expressions that call functions. Recursion is possible, up to a nesting depth limit. Figure 6 displays function syntax.

```
# Red Hat convention; see /etc/login.defs UID_MIN
function system_uid_p (u) { return u < 500 }

# kernel device number assembly macro
function makedev (major,minor) { return major << 20 | minor }

function trace_common ()
{
  printf("%d %s(%d)", gettimeofday_s(), execname(), pid())
  # no return value necessary
}

function fibonacci (i)
{
  if (i < 1) return 0
  else if (i < 2) return 1
  else return fibonacci(i-1) + fibonacci(i-2)
}
```

Figure 6: Some functions of dubious utility.

3.4 Arrays

Often, probes will want to share data that cannot be represented as a simple scalar value. Much data is naturally tabular in nature, indexed by some tuple of thread numbers, processor ids, names, time, and so on. Systemtap offers associative arrays for this purpose. These arrays are implemented as hash tables with a maximum size that is fixed at startup. Because they are too large to be created dynamically for individual probes handler runs, they must be declared as global.

```
      global a   declare global scalar or array variable
global b[400]    declare array, reserving space for up to 400 tuples
```

The basic operations for arrays are setting and looking up elements. These are expressed in `awk` syntax: the array name followed by an opening `[` bracket, a comma-separated list of index expressions, and a closing `]` bracket. Each index expression may be string or numeric, as long as it is consistently typed throughout the script.

```
                  foo [4,"hello"] ++    increment the named array slot
  processusage [uid(),execname()] ++    update a statistic
        times [tid()] = get_cycles()    set a timestamp reference point
delta = get_cycles() - times [tid()]    compute a timestamp delta
```

Array elements that have not been set *may* be fetched, and return a dummy null value (zero or an empty string) as appropriate. However, assigning a null value does not delete the element: an explicit `delete` statement is required. Systemtap provides syntactic sugar for these operations, in the form of explicit membership testing and deletion.

```
if ([4,"hello"] in foo) { }   membership test
        delete times[tid()]   deletion of a single element
               delete times   deletion of all elements
```

One final and important operation is iteration over arrays. This uses the keyword `foreach`. Like `awk`, this creates a loop that *iterates over key tuples* of an array, not just *values*. In addition, the iteration may be *sorted* by any single key or the value by adding an extra + or − code.

The `break` and `continue` statements work inside `foreach` loops, too. Since arrays can be large but probe handlers must not run for long, it is a good idea to exit iteration early if possible. The `limit` option in the `foreach` expression is one way. For simplicity, systemtap forbids any *modification* of an array while it is being iterated using a `foreach`.

```
foreach (x = [a,b] in foo) { fuss_with(x) }   simple loop in arbitrary sequence
        foreach ([a,b] in foo+ limit 5) { }   loop in increasing sequence of value, stop
                                              after 5
                foreach ([a-,b] in foo) { }   loop in decreasing sequence of first key
```

3.5 Aggregates

When we said above that values can only be strings or numbers, we lied a little. There is a third type: statistics aggregates, or aggregates for short. Instances of this type are used to collect statistics on numerical values, where it is important to accumulate new data quickly (*without* exclusive locks) and in large volume (storing only aggregated stream statistics). This type only makes sense for global variables, and may be stored individually or as elements of an array.

To add a value to a statistics aggregate, systemtap uses the special operator <<<. Think of it like C++'s << output streamer: the left hand side object accumulates the data sample given on the right hand side. This operation is efficient (taking a shared lock) because the aggregate values are kept separately on each

processor, and are only aggregated across processors on request.

```
a <<< delta_timestamp
writes[execname()] <<< count
```

To read the aggregate value, special functions are available to extract a selected statistical function. *The aggregate value cannot be read by simply naming it as if it were an ordinary variable.* These operations take an exclusive lock on the respective globals, and should therefore be relatively rare. The simple ones are: `@min`, `@max`, `@count`, `@avg`, and `@sum`, and evaluate to a single number. In addition, histograms of the data stream may be extracted using the `@hist_log` and `@hist_linear`. These evaluate to a special sort of array that may at present[3] only be printed.

`@avg(a)`	the average of all the values accumulated into `a`
`print(@hist_linear(a,0,100,10))`	print an "ascii art" linear histogram of the same data stream, bounds $0 \ldots 100$, bucket width is 10
`@count(writes["zsh"])`	the number of times "zsh" ran the probe handler
`print(@hist_log(writes["zsh"]))`	print an "ascii art" logarithmic histogram of the same data stream

3.6 Safety

The full expressivity of the scripting language raises good questions of safety. Here is a set of Q&A:

What about infinite loops? recursion? A probe handler is bounded in time. The C code generated by systemtap includes explicit checks that limit the total number of statements executed to a small number. A similar limit is imposed on the nesting depth of function calls. When either limit is exceeded, that probe handler cleanly aborts and signals an error. The systemtap session is normally configured to abort as a whole at that time.

What about running out of memory? No dynamic memory allocation whatsoever takes place during the execution of probe handlers. Arrays, function contexts, and buffers are allocated during initialization. These resources may run out during a session, and generally result in errors.

What about locking? If multiple probes seek conflicting locks on the same global variables, one or more of them will time out, and be aborted. Such events are tallied as "skipped" probes, and a count is displayed at session end. A configurable number of skipped probes can trigger an abort of the session.

What about null pointers? division by zero? The C code generated by systemtap translates potentially dangerous operations to routines that check their arguments at run time. These signal errors if they are invalid. Many arithmetic and string operations silently overflow if the results exceed representation limits.

What about bugs in the translator? compiler? While bugs in the translator, or the runtime layer certainly exist[4], our test suite gives some assurance. Plus, the entire generated C code may be inspected (try the `-p3` option). Compiler bugs are unlikely to be of any greater concern for systemtap than for the kernel as a whole. In other words, if it was reliable enough to build the kernel, it will build the systemtap modules properly too.

Is that the whole truth? In practice, there are several weak points in systemtap and the underlying kprobes system at the time of writing. Putting probes indiscriminately into unusually sensitive parts

[3] We anticipate support for indexing and looping using `foreach` shortly.
[4] See `http://sourceware.org/bugzilla`

of the kernel (low level context switching, interrupt dispatching) has reportedly caused crashes in the past. We are fixing these bugs as they are found, and constructing a probe point "blacklist", but it is not complete.

3.7 Exercises

1. Alter the last probe in `timer-jiffies.stp` to reset the counters and continue reporting instead of exiting.
2. Write a script that, every ten seconds, displays the top five most frequent users of `open` system call during that interval.
3. Write a script that experimentally measures the speed of the `get_cycles()` counter on each processor.
4. Use any suitable probe point to get an approximate profile of process CPU usage: which processes/users use how much of each CPU.

4 Tapsets

After writing enough analysis scripts for yourself, you may become known as an expert to your colleagues, who will want to use your scripts. Systemtap makes it possible to share in a controlled manner; to build libraries of scripts that build on each other. In fact, all of the functions (`pid()`, etc.) used in the scripts above come from tapset scripts like that. A "tapset" is just a script that designed for reuse by installation into a special directory.

4.1 Automatic selection

Systemtap attempts to resolve references to global symbols (probes, functions, variables) that are not defined within the script by a systematic search through the tapset library for scripts that define those symbols. Tapset scripts are installed under the default directory named `/usr/share/systemtap/tapset`. A user may give additional directories with the `-I DIR` option. Systemtap searches these directories for script (`.stp`) files.

The search process includes subdirectories that are specialized for a particular kernel version and/or architecture, and ones that name only larger kernel families. Naturally, the search is ordered from specific to general, as shown in Figure 7.

When a script file is found that *defines* one of the undefined symbols, that *entire file* is added to the probing session being analyzed. This search is repeated until no more references can become satisfied. Systemtap signals an error if any are still unresolved.

This mechanism enables several programming idioms. First, it allows some global symbols to be defined only for applicable kernel version/architecture pairs, and cause an error if their use is attempted on an inapplicable host. Similarly, the same symbol can be defined differently depending on kernels, in much the same way that different kernel `include/asm/ARCH/` files contain macros that provide a porting layer.

Another use is to separate the default parameters of a tapset routine from its implementation. For example, consider a tapset that defines code for relating elapsed time intervals to process scheduling activities. The data collection code can be generic with respect to which time unit (jiffies, wall-clock seconds, cycle counts) it can use. It should have a default, but should not require additional run-time checks to let a user choose another. Figure 8 shows a way.

```
# stap -p1 -vv -e 'probe begin { }' > /dev/null
Created temporary directory "/tmp/staplnEBh7"
Searched '/usr/share/systemtap/tapset/2.6.15/i686/*.stp', match count 0
Searched '/usr/share/systemtap/tapset/2.6.15/*.stp', match count 0
Searched '/usr/share/systemtap/tapset/2.6/i686/*.stp', match count 0
Searched '/usr/share/systemtap/tapset/2.6/*.stp', match count 0
Searched '/usr/share/systemtap/tapset/i686/*.stp', match count 1
Searched '/usr/share/systemtap/tapset/*.stp', match count 12
Pass 1: parsed user script and 13 library script(s) in 350usr/10sys/375real ms.
Running rm -rf /tmp/staplnEBh7
```

Figure 7: Listing the tapset search path.

```
# cat tapset/time-common.stp
global __time_vars
function timer_begin (name) { __time_vars[name] = __time_value () }
function timer_end (name) { return __time_value() - __time_vars[name] }

# cat tapset/time-default.stp
function __time_value () { return gettimeofday_us () }

# cat tapset-time-user.stp
probe begin
{
  timer_begin ("bench")
  for (i=0; i<100; i++) ;
  printf ("%d cycles\n", timer_end ("bench"))
  exit ()
}
function __time_value () { return get_ticks () } # override for greater precision
```

Figure 8: Providing an overrideable default.

A tapset that exports only *data* may be as useful as ones that exports functions or probe point aliases (see below). Such global data can be computed and kept up-to-date using probes internal to the tapset. Any outside reference to the global variable would incidentally activate all the required probes.

4.2 Probe point aliases

Probe point aliases allow creation of new probe points from existing ones. This is useful if the new probe points are named to provide a higher level of abstraction. For example, the system-calls tapset defines probe point aliases of the form `syscall.open` etc., in terms of lower level ones like `kernel.function("sys_open")`. Even if some future kernel renames `sys_open`, the aliased name can remain valid.

A probe point alias definition looks like a normal probe. Both start with the keyword `probe` and have a probe handler statement block at the end. But where a normal probe just lists its probe points, an alias creates a new name using the assignment (=) operator. Another probe that names the new probe point will create an actual probe, with the handler of the alias *prepended*.

This prepending behavior serves several purposes. It allows the alias definition to "preprocess" the context of the probe before passing control to the user-specified handler. This has several possible uses:

`if ($flag1 != $flag2) next`	skip probe unless given condition is met
`name = "foo"`	supply probe-describing values
`var = $var`	extract target variable to plain local variable

Figure 9 demonstrates a probe point alias definition as well as its use. It demonstrates how a single probe point alias can expand to multiple probe points, even to other aliases. It also includes probe point wildcarding. These functions are designed to compose sensibly.

```
# cat probe-alias.stp
probe syscallgroup.io = syscall.open, syscall.close,
                        syscall.read, syscall.write
{ groupname = "io" }

probe syscallgroup.process = syscall.fork, syscall.execve
{ groupname = "process" }

probe syscallgroup.*
{ groups [execname() . "/" . groupname] ++ }

probe end
{
  foreach (eg+ in groups)
    printf ("%s: %d\n", eg, groups[eg])
}

global groups

# stap probe-alias.stp
05-wait_for_sys/io: 19
10-udev.hotplug/io: 17
20-hal.hotplug/io: 12
X/io: 73
apcsmart/io: 59
[...]
make/io: 515
make/process: 16
[...]
xfce-mcs-manage/io: 3
xfdesktop/io: 5
[...]
xmms/io: 7070
zsh/io: 78
zsh/process: 5
```

Figure 9: Classified system call activity.

4.3 Embedded C

Sometimes, a tapset needs provide data values from the kernel that cannot be extracted using ordinary target variables (`$var`). This may be because the values are in complicated data structures, may require lock awareness, or are defined by layers of macros. Systemtap provides an "escape hatch" to go beyond what the language can safely offer. In certain contexts, you may embed plain raw C in tapsets, exchanging power for the safety guarantees listed in section 3.6. End-user scripts *may not* include embedded C code, unless systemtap is run with the `-g` ("guru" mode) option. Tapset scripts get guru mode privileges automatically.

Embedded C can be the body of a script function. Instead enclosing the function body statements in `{` and `}`, use `%{` and `%}`. Any enclosed C code is literally transcribed into the kernel module: it is up to you to make it safe and correct. In order to take parameters and return a value, macros `STAP_ARG_*` and `STAP_RETVALUE` are made available. The familiar data-gathering functions `pid()`, `execname()`, and their neighbours are all embedded C functions. Figure 10 contains another example.

Since systemtap cannot examine the C code to infer these types, an optional[5] annotation syntax is available to assist the type inference process. Simply suffix parameter names and/or the function name with `:string` or `:long` to designate the string or numeric type. In addition, the script may include a `%{ %}` block at the outermost level of the script, in order to transcribe declarative code like `#include <linux/foo.h>`. These enable the embedded C functions to refer to general kernel types.

There are a number of safety-related constraints that should be observed by developers of embedded C code.

1. Do not dereference pointers that are not known or testable valid.
2. Do not call any kernel routine that may cause a sleep or fault.
3. Consider possible undesirable recursion, where your embedded C function calls a routine that may be the subject of a probe. If that probe handler calls your embedded C function, you may suffer infinite regress. Similar problems may arise with respect to non-reentrant locks.
4. If locking of a data structure is necessary, use a `trylock` type call to attempt to take the lock. If that fails, give up, do not block.

4.4 Naming conventions

Using the tapset search mechanism just described, potentially many script files can become selected for inclusion in a single session. This raises the problem of name collisions, where different tapsets accidentally use the same names for functions/globals. This can result in errors at translate or run time.

To control this problem, systemtap tapset developers are advised to follow naming conventions. Here is some of the guidance.

1. Pick a unique name for your tapset, and substitute it for *TAPSET* below.
2. Separate identifiers meant to be used by tapset users from those that are internal implementation artifacts.
3. Document the first set in the appropriate `man` pages.
4. Prefix the names of external identifiers with *TAPSET*_ if there is any likelihood of collision with other tapsets or end-user scripts.

[5] This is only necessary if the types cannot be inferred from other sources, such as the call sites.

```
# cat embedded-C.stp
%{
#include <linux/sched.h>
#include <linux/list.h>
%}

function task_execname_by_pid:string (pid:long) %{
  struct task_struct *p;
  struct list_head *_p, *_n;
  list_for_each_safe(_p, _n, &current->tasks) {
    p = list_entry(_p, struct task_struct, tasks);
    if (p->pid == (int)STAP_ARG_pid)
      snprintf(STAP_RETVALUE, MAXSTRINGLEN, "%s", p->comm);
  }
%}

probe begin
{
  printf("%s(%d)\n", task_execname_by_pid(target()), target())
  exit()
}

# pgrep emacs
16641
# stap -g embedded-C.stp -x 16641
emacs(16641)
```

Figure 10: Embedded C function.

5. Prefix any probe point aliases with an appropriate prefix.
6. Prefix the names of internal identifiers with *__TAPSET_*.

4.5 Exercises

1. Write a tapset that implements deferred and "cancelable" logging. Export a function that enqueues a text string (into some private array), returning an id token. Include a timer-based probe that periodically flushes the array to the standard log output. Export another function that, if the entry was not already flushed, allows a text string to be cancelled from the queue. One might speculate that similar functions and tapsets exist.
2. Create a "relative timestamp" tapset with functions return all the same values as the ones in the timestamp tapset, except that they are made relative to the start time of the script.
3. Create a tapset that exports a global array that contains a mapping of recently seen process ID numbers to process names. Intercept key system calls (`execve`?) to update the list incrementally.
4. Send your tapset ideas to the mailing list!

5 Further information

For further information about systemtap, several sources are available.

There are `man` pages:

`stap`	systemtap program usage, language summary
`stappaths`	your systemtap installation paths
`stapprobes`	probes / probe aliases provided by built-in tapsets
`stapex`	a few basic example scripts
`tapset::*`	summaries of the probes and functions in each tapset
`probe::*`	detailed descriptions of each probe
`function::*`	detailed descriptions of each function

There is much more documentation and sample scripts included. You may find them under `/usr/share/doc/systemtap*/`.

Then, there is the source code itself. Since systemtap is *free software*, you should have available the entire source code. The source files in the `tapset/` directory are also packaged along with the systemtap binary. Since systemtap reads these files rather than their documentation, they are the most reliable way to see what's inside all the tapsets. Use the `-v` (verbose) command line option, several times if you like, to show inner workings.

Finally, there is the project web site (`http://sourceware.org/systemtap/`) with several articles, an archived public mailing list for users and developers (`systemtap@sourceware.org`), IRC channels, and a live GIT source repository. Come join us!

A Glossary

Sorry, no glossary available.

B Errors

We explain some common systemtap error messages in this section. Most error messages include line/character numbers with which one can locate the precise location of error in the script code. There is sometimes a subsequent or prior line that elaborates.

error `at:` *filename*:*line*:*column*: *details*

B.1 Parse errors

parse error: expected *foo*, saw *bar* ...
The script contained a grammar error. A different type of construct was expected in the given context.

parse error: embedded code in unprivileged script
The script contained unsafe constructs such as embedded C (section 4.3), but was run without the `-g` (guru mode) option. Confirm that the constructs are used safely, then try again with `-g`.

B.2 Type errors

semantic error: type mismatch for identifier '*foo*' ... string vs. long

In this case, the identifier *foo* was previously inferred as a numeric type ("long"), but at the given point is being used as a string. Similar messages appear if an array index or function parameter slot is used with conflicting types.

semantic error: unresolved type for identifier *'foo'*
The identifier *foo* was used, for example in a `print`, but without any operations that could assign it a type. Similar messages may appear if a symbol is misspelled by a typo.

semantic error: Expecting symbol or array index expression
Something other than an assignable lvalue was on the left hand sign of an assignment.

B.3 Symbol errors

while searching for arity *N* function, semantic error: unresolved function call
The script calls a function with *N* arguments that does not exist. The function may exist with different arity.

semantic error: array locals not supported: ...
An array operation is present for which no matching global declaration was found. Similar messages appear if an array is used with inconsistent arities.

semantic error: variable *'foo'* modified during 'foreach'
The array *foo* is being modified (being assigned to or deleted from) within an active `foreach` loop. This invalid operation is also detected within a function called from within the loop.

B.4 Probing errors

semantic error: probe point mismatch at position *N*, while resolving probe point *foo*
A probe point was named that neither directly understood by systemtap, nor defined as an alias by a tapset script. The divergence from the "tree" of probe point namespace is at position *N* (starting with zero at left).

semantic error: no match for probe point, while resolving probe point *foo*
A probe point cannot be resolved for any of a variety of reasons. It may be a debuginfo-based probe point such as `kernel.function("foobar")` where no `foobar` function was found. This can occur if the script specifies a wildcard on function names, or an invalid file name or source line number.

semantic error: unresolved target-symbol expression
A target variable was referred to in a probe handler that was not resolvable. Or, a target variable is not valid at all in a context such as a script function. This variable may have been elided by an optimizing compiler, or may not have a suitable type, or there might just be an annoying bug somewhere. Try again with a slightly different probe point (use `statement()` instead of `function()`) to search for a more cooperative neighbour in the same area.

semantic error: libdwfl failure ...
There was a problem processing the debugging information. It may simply be missing, or may have some consistency / correctness problems. Later compilers tend to produce better debugging information, so if you can upgrade and recompile your kernel/application, it may help.

semantic error: cannot find *foo* debuginfo
Similarly, suitable debugging information was not found. Check that your kernel build/installation includes a matching version of debugging data.

B.5 Runtime errors

Usually, run-time errors cause a script to terminate. Some of these may be caught with the `try { ... } catch { ... }` construct.

WARNING: Number of errors: *N*, skipped probes: *M*
Errors and/or skipped probes occurred during this run.

division by 0
The script code performed an invalid division.

aggregate element not found
An statistics extractor function other than `@count` was invoked on an aggregate that has not had any values accumulated yet. This is similar to a division by zero.

aggregation overflow
An array containing aggregate values contains too many distinct key tuples at this time.

MAXNESTING exceeded
Too many levels of function call nesting were attempted.

MAXACTION exceeded
The probe handler attempted to execute too many statements.

kernel/user string copy fault at *0xaddr*
The probe handler attempted to copy a string from kernel or user space at an invalid address.

pointer dereference fault
There was a fault encountered during a pointer dereference operation such as a target variable evaluation.

C Acknowledgments

The author thanks Martin Hunt, Will Cohen, and Jim Keniston for improvement advice for this paper.