# Introduction to Systemtap

**Ulrich Drepper**
**Consulting Engineer**

redhat

# What is the Problem?

- Examine live systems
  - No preparation
  - Reversible operation
  - Easy writing of code to perform examination
  - Ability to run one piece of code after the other

redhat

# What was Available?

- oprofile
  - Test for reachability
  - Statistical profiling
- Kernel debugger
  - Disruptive
  - Only examination and state manipulation
- kprobes
  - Infrastructure for kernel code to inject code into functions
  - Injected code usually in kernel modules

redhat

# Sun made the leap

- dtrace
  - Inject code into kernel and userlevel code
  - Code written in easy-to-learn scripting language
  - Sun loves byte interpreters:
    - Scripts translated into byte code
  - Documented insertion points

redhat

# ... but there are limitations

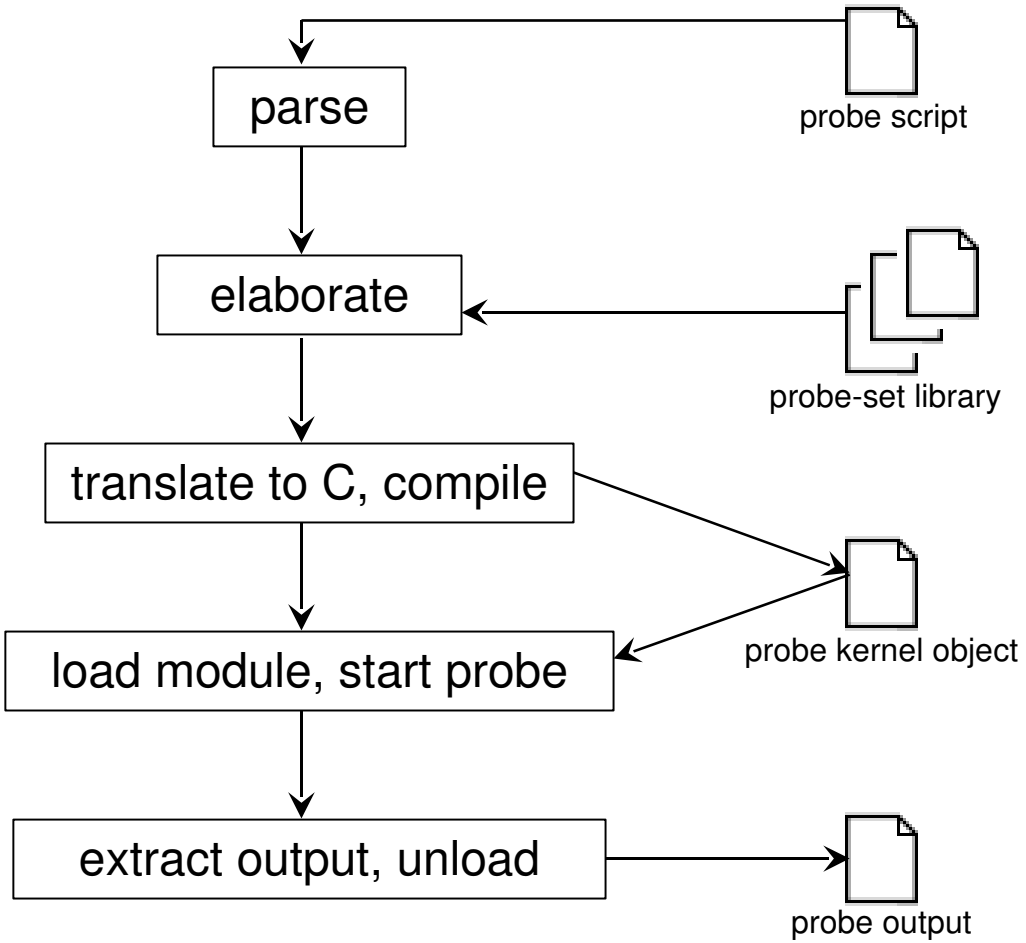
- Byte code interpreter much slower than native code
  - Limits amount of work scripts can do without impacting system
  - Complicated piece of code itself
- Fixed insertion points
  - You can do exactly what Sun thinks you should be able to do
- No generic monitoring
  - Cannot access arbitrary variables

redhat

# A Different Approach: Systemtap

- More powerful scripting language
  - Arbitrary insertion points
  - Use all variables in the code
- Compiled code instead of byte code
- Lowlevel language with extensible collection of library functions
- Build on top of kprobes (and now other, similar technology)

# Systemtap Workflow



redhat

# Specifying Probes

- Basic syntax:

```
probe name {
   body
}
```

- Possibilities for ***name***
  - **`start`** and **`end`** for initializers and finalizers respectively
  - **`kernel.function("`*name*`")`** for beginning of function ***name***
  - **`kernel.function("`*name*`").return`** for end of function ***name***
  - **`statement(`*addr*`)`** at given address ***addr***
  - **`module("`*name*`").`*name*** recursively in loadable module ***name***
  - **`timer.ms(`*value*`)`** for interval timer with given interval length

redhat

# Specifying Probes

- Possibilities for ***body***
  - **`if (`***`expr`***`)`** ***body*** **`else`** ***body***
  - **`while (expr)`** ***body***
  - **`for (`***`A`***`; `***`B`***`; `***`C`***`)`** ***body***
  - **`break`**, **`continue`**, **`return`** ***val***
  - Variables, associative arrays
  - **`foreach (`***`var`* **`in`** ***`array`***`)`** ***body*** to iterate over array content
  - Target variables **`$`***`var`*
  - Access to system values like **`tid()`**, **`pid()`**, **`uid()`**, **`execname()`**, **`get_cycles()`**, and many more

redhat

# Specifying Probes

- Global variables
  - **`global`** ***name***
- Printing
  - **`print`**, **`sprint`** to print single values
  - **`printf`**, **`sprintf`** to print formatted

- Statistics
  - ***var*** **`<<<`** ***value*** adds ***value*** to the statistic set ***var***
  - Extractors
    - **`@count`**, **`@sum`**, **`@min`**, **`@max`**, **`@avg`**
    - ASCII Art printing
      - **`@hist_linear`**, **`@hist_log`**

redhat

# First Example

- Notify when entering and leaving function to open a file:

```
global assoc
probe kernel.function("do_sys_open") {
  assoc[tid()] = user_string($filename)
  printf("enter %s for %s\n", probefunc(),
         assoc[tid()])
}
probe kernel.function("do_sys_open").return {
  printf("leave %s for %s\n", probefunc(),
         assoc[tid()])
}
```

redhat

# Problem With This Approach

**The programmer needs to know too many details!**

- Names of functions in kernel sources
- Worse: name of function parameters
- Collecting parameter list for printing tiresome
- Same for return value

redhat

# Enter: Tapscripts

- Written by the people who ought to know the details
  - Distributed along with systemtap for generic kernel functions
  - Can be distributed with kernel modules
  - Separate collections by people with domain knowledge
  - All collections complement each other
- Remove requirement for detailed knowledge
- Summarize activities

redhat

# First Example (revisited)

- Notify when entering and leaving function to open a file:

```
global assoc
probe syscall.open {
  assoc[tid()] = argstr
  printf("enter %s for %s\n", probefunc(),
          assoc[tid()])
}
probe syscall.open.return {
  printf("leave %s for %s\n", probefunc(),
          assoc[tid()])
}
```

redhat

# How It's Done

- The definition from the standard tapset library:

```
probe syscall.open =
                kernel.function("sys_open") ?,
                kernel.function("compat_sys_open") ?,
                kernel.function("sys32_open") ?
{
    name = "open"
    filename = user_string($filename)
    flags = $flags
    mode = $mode
    if (flags & 64)
        argstr = sprintf("%s, %s, %#o", user_string_quoted($filename),
                         _sys_open_flag_str(flags),$mode)
    else
        argstr = sprintf("%s, %s", user_string_quoted($filename),
                         _sys_open_flag_str(flags))
}
```

redhat

# Probes to Abstract and Refine

- Extended probe syntax

```
probe name = name [?], name [?], [...] {
    body
}
```

- and

```
probe name += name [?], name [?], [...] {
    body
}
```

redhat

# Make it more usable

- Probes can be big
- Just as everywhere else: code should be reused
- Functions

```
function name[:type](arg1, arg2, [...]) {
    body
}
```

- Large collection of functions in standard runtime
  - Data extraction (e.g., **`struct timeval`**)
  - Diagnostic message (e.g., backtrace, register content)
  - Value handling (e.g., byte order conversion)
  - Access kernel data structures (e.g., **`tid()`**, extract IP address from socket)
- See **`man stapfuncs`**

redhat

# Allow Library to be Generic

- Kernel details differ
  - Between different versions
  - Between different architectures
- Minimal preprocessor support

```
%( kernel_v >= "2.6.10" %?
    code
%:
    code
%)
```

redhat

# Only for Gurus

- Possible to insert C code
- Syntax similar to normal functions
- Only allowed if -g flag added to command line of stap

```
function add_one(val) %{
     THIS->__retvalue = THIS->val + 1;
%}
function add_one_str(val) %{
  strlcpy(THIS->__retvalue, THIS->val, MAXSTRINGLEN);
  strlcat(THIS->__retvalue, "one", MAXSTRINGLEN);
%}
```

redhat

# Available High-Level Tapscripts

- I/O: scheduler, system call, AIO
- Network device handling
- NFS, RPC
- SCSI
- Process handling
- Memory handling (page faults etc)
- Signals

Use **`man stapprobe`**

redhat

# Using stap

stap options

- -v increase verbosity to diagnose problems
- -g Guru mode
- -p NUM stop after pass NUM (parse, elaborate, translate, compile, run)
- -I DIR look for additional .stp scripts
- -r REL cross-compile for kernel version REL
- -o FILE send output into FILE
- -x PID sets `target()` to PID
- -c CMD start CMD and terminate when it exits

redhat

# Tips for Writing Tapscripts

- Kernel memory is precious, especially on life systems
  - Data structures like arrays can grow
- Transport of output to userlevel not free
  - relayfs can handle traffic
  - Per-cpu buffers
  - But: delays in reader process might stall everything

- Need to find a balance between keeping data in kernel and sending to userlevel
  - Caching data for seconds/minutes, then sending summary
    - For instance, `probe timer.ms(5000) { ...; delete array }`
  - Sending binary data which needs to be post-processed

redhat

# Question?

http://sources.redhat.com/systemtap/