IBM Beaverton Linux Change Team

# Using SystemTap with Linux on System z

Mike O'Reilly

# SystemTap

- Scripting language and tools
  - dynamic tracing/probing
    - kernel functions
    - system calls
    - Kernel-space events
    - User-space events ( newest versions )
  - Identifying the underlying cause of a bug
  - Performance problems

- Eliminate instrument, recompile, install, and reboot sequence

# Installing SystemTap – RedHat

- systemtap
- systemtap-runtime

- kernel-debuginfo
- kernel-debuginfo-common-arch
- kernel-devel

- Test
  - stap -v -e 'probe vfs.read {printf("  read performed\n"); exit()}'

Pass 1: parsed user script …
…
Pass 5: starting run.
  Read performed
Pass 5: run completed in 10usr/70sys/423 real ms.

# Installing SystemTap - Novell

- systemtap

- Kernel-source
- Kernel-default-debuginfo
- Development packages
- ** kernel build environment

- Test
  - stap -v -e 'probe vfs.do_sync_read {printf(“read performed\n”); exit()}'

# SystemTap's scripting language

- Procedural
- C-like
- Integers, Strings, Associative arrays, Statistics aggregates
- Implicitly typed

- Based on Two main function constructs:
  - Probes
  - Functions
    - statements and expressions use C-like operator syntax and precedence

# Primary construct: probe

probe <event> { handler }

- event is
  - kernel.function,
  - process.statement,
  - timer.ms,
  - begin, end (tapset aliases).
- handler can have:
  - variables
  - filtering/conditionals (if ... next)
  - control structures (foreach, while)

# Probe example

\# cat simple.stp

```
#!/usr/bin/stap
probe begin {printf("Probe started\n");}
probe timer.sec(3) {exit();}
probe end {printf("Probe ended\n");}
```

Events: begin, timer.sec, end
Handlers: printf(), exit()

# probe example cont.

```
# stap simple.stp
        Probe started
          … 3 seconds later ...
        Probe ended
```

The stap program

- The front-end to the SystemTap tool.
  - Translates the script into C code
  - Compiles C and Generates a kernel module
  - Inserts the module;
  - Output to stap's stdout

CTRL-C unloads the module, terminates stap

# probe example

# cat sigaltstack.stp

```
probe kernel.function("sys_sigaltstack") {
    printf("sys_sigaltstack called\n"); }
```

- Event: kernel.function("sys_sigaltstack")
- Handler: { printf("sys_sigaltstack called\n"); }

To specify the return of the kernel function

- Event: kernel.function("sys_sigaltstack").return

# probe Event examples

- syscall.read

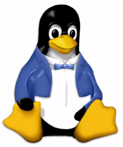
  when entering read() system call
- syscall.close.return

  when returning from the close() system call
- module("*dasd*").function("*")

  when entering any function in the "dasd" module
- kernel.function("*@net/socket.c").return

  returning from any function in file net/socket.c
- kernel.statement("*@kernel/sched.c:2917")

  when hitting line 2917 of file kernel/sched.c



# Probe Event ex. cont.

```
# cat dasd_callgraph.stp
probe module("*dasd*").function("*@drivers/s390/block/dasd.c").call {
  printf ("%s -> %s\n", thread_indent(1), probefunc()) }
probe module("*dasd*").function("*@drivers/s390/block/dasd.c").return {
  printf ("%s <- %s\n", thread_indent(-1), probefunc()) }

# stap dasd_callgraph.stp
     0 bash(34989): -> dasd_generic_set_offline
    80 bash(34989):  -> dasd_set_target_state
   100 bash(34989):   -> dasd_change_state
   336 bash(34989):    -> dasd_flush_block_queue
   357 bash(34989):    <- dasd_flush_block_queue
 21898 bash(34989):    -> dasd_release
 21925 bash(34989):    <- dasd_release
 53615 bash(34989):    -> dasd_block_clear_timer
 53640 bash(34989):    <- dasd_block_clear_timer
```

# probe Events cont.

- timer.ms(200)

  every 200 milliseconds
- process("/bin/ls").function("*")

  entering any function in /bin/ls (not its libraries or syscalls)
- process("/lib/libc.so.6").function("*malloc*")

  entering any glibc function with "malloc" in its name
- kernel.function("*init*"),
- kernel.function("*exit*").return

  entering any kernel function which has "init" in its name or returning from any kernel function with "exit" in its name

# probe Events cont.

- Optional probes
  - kernel.function(“may not exist”) ? { ... }

  - kernel.function(“this might exist”) !,
  - kernel.function(“if not then this should”) !,
  - kernel.function(“if all else fails”) { ... }
- Conditional probes
  - probe kernel.function("some func") if ( val > 10)
- Filter
  - stap -e -x PID 'probe syscall.* { if (pid() == target()) printf(“%s\n”,name)}

# Handler Constructs

- Variables
  - Script
  - Target
- Conditional Statements
- Loops

# Variables

## Script variables

- Global or local
- Automatically typed: type inferred from assignment
  - Integers (64-bit signed)
  - Strings
  - Associative arrays (global only),
  - Statistics aggregates (global only)
- Automatically initialized to zero/empty

# Variables example

```
# cat vars.stp
global x=64
global arr

probe begin {

   i = 10
   name = "Mike"

   arr[0] = 1
   arr[2] = 3
   printf(" x: %d\n i: %d\n name: %s\n", x,i,name)
   foreach( y in arr ) {
      printf("y: %d arr[y]: %d\n", y, arr[y])
   }
   exit()
}

# stap vars.stp
 x: 64
 i: 10
 name: Mike
y: 0 arr[y]: 1
y: 2 arr[y]: 3
```

# Variables – Array example

```
# cat pfaultByProcess.stp
global numFaults

probe vm.pagefault{ numFaults[ execname() ] += 1 }

probe timer.s(5) {
   printf ("%16s\t%10s\t\n", "Process", "Num pagefaults")
   foreach (name in numFaults- ) {
      printf ("%16s\t%d\n", name, numFaults[name] ) }
   exit() }

# stap pfaultByProtcess.stp
         Process   Num pagefaults
              ps          300
            bash           67
          stapio           10
```

# Variables – Stat Aggregate example

```
global NumReads
probe vfs.read { NumReads[execname()] <<< $count }
probe timer.s(5) {
  foreach (name in NumReads ) {
    printf ("%16s\t%d\n", name, @count(NumReads[name]) ) }
  foreach (name in NumReads ) {
    printf ("%16s\t%d\n", name, @sum(NumReads[name]) ) }
  exit() }
```

```
 Process      Number reads
     crond          4
   rsyslogd         1
 Process      Total Bytes read
     crond       16384
   rsyslogd       4095
```

# Variables Cont.

## Target variables

- Variables defined in the source code at event location

```
int qeth_setup_channel(struct qeth_channel* channel) {
  int cnt;
$channel, $cnt
```

- special variables – e.g., $return, $$parms, $$vars
- For pointers to base types such as integers and strings
  - kernel_long(address), kernel_string(address) for safe access to variable values.

```
# stap -L 'module("*dasd*").function("dasd_alloc_queue")'
module("dasd_mod").function("dasd_alloc_queue@drivers/s390/block/dasd.c:2182") $block:struct dasd_block*
```

# Variables Cont.

```
# stap -e 'probe module("*dasd*").function("dasd_alloc_queue")
                                   {printf("%s\n", $$parms); exit(); }'
block=0x3eee3800
```

"$" suffix to pretty print the data structure.

```
# stap -e 'probe module("*dasd*").function("dasd_alloc_queue")
                          {printf("%s\n", $$parms$); exit(); }'
block={.gdp=0x0, .request_queue=0x0, ...,  .base=0x3eec3400,
.ccw_queue={...}, .queue_lock={...}, ...}
```

"$$" suffix will print the values within the nested data structures

```
# stap -e 'probe module("*dasd*").function("dasd_alloc_queue")
                          {printf("%s\n", $$parms$$); exit(); }'
block={.gdp=0x0, .request_queue=0x0 ,...,
.ccw_queue={.next=0x3f138840, .prev=0x3f138840},
.queue_lock={.raw_lock={.owner_cpu=0}},...
```

# Variables Cont.

```
# stap -L 'kernel.function("sys_sigaltstack")'
kernel.function("SyS_sigaltstack@arch/s390/kernel/signal.c:106") $uss:long int
$uoss:long int
uss   (const stack_t *)  points to a signalstack structure

# cat sigaltstack.stp
probe kernel.function("sys_sigaltstack") { printf( "%s\n", $$parms$ );}

# stap sigaltstack.stp
uss=2102012640 uoss=0
uss=2102029024 uoss=0
uss=2144010128 uoss=0
uss=4302019688 uoss=0
```

# Script and Target variables

```
global openFails, huge_reads
probe kernel.function("sys_open").return {
  if ($return < 0) openFails++;
}
probe kernel.function("sys_read") {
  if ($count > 4*1024) huge_reads++;
}
```

- Script variables: openFails, huge_reads
- Target variable: $count – sys_read()'s 3rd arg
- Special context variable: $return

# Conditional/Loop statements

- Group compound statements with { }
- Branching
  - if (condition) statement1 [else statement2]
- Looping:
  - while (condition) statement
  - for (initial; condition; iteration) statement
  - foreach ([VAR1, VAR2] in ARRAY [limit NUM]) statement
  - break; continue;
- Other:
  - return [VAL]; next; delete VAR;

# Conditional/Loop Statements Example

```
if (flag & CREAT_FLAG) return 1
else return 0

for(i=0;i<10;i++) { ... }

while (i<10) { ... }

foreach (item in myarr) { myarr[item]++ }
```

# Primary construct: function

```
function NAME:type(ARG1:type, ARG2:type) {
    /* code to run when NAME is called */

  return VALUE
}
```

- The :types are optional, and may be
  - :string
  - :long.

# Example: function

```
# cat func.stp
function is_open_creating:long (flag:long){
  CREAT_FLAG = 4 // 0x4 = 00000100b
  if (flag & CREAT_FLAG) return 1
  else return 0
}
probe kernel.function("sys_open"){
   creating = is_open_creating($mode)
   if (creating)
      printf("Creating file %s\n", user_string($filename))
   else
      printf("Opening file %s\n", user_string($filename))
}

# stap func.stp
Opening file public/pickup
Opening file maildrop
Opening file /lib64/libwrap.so.0
Creating file /etc/selinux/config
Creating file /proc/mounts
...
```

# Useful helper functions

- pid() - which process is this?
- uid() - which user is running this?
- execname() - what is the name of this process?
- tid() - which thread is this?
- gettimeofdays() - epoch time in seconds
- probefunc() - what function are we in?
- print_backtrace() - print stack back trace

See “man stapfuncs” for details and many more

# Tapsets - pre-written probe libraries

- Tapsets
  - provide easy to use aliases of common probepoints,
  - provide values of interest from those probepoints,
  - define those helper functions
- Tapsets are SystemTap scripts.
  - not runnable (probe aliases, not probes)
  - installed in /usr/share/systemtap/tapset/
- Typically encapsulate knowledge about a particular application or kernel subsystem

# Example of a tapset function

```
probe vm.pagefault = kernel.function("handle_mm_fault")
{
        name = "pagefault"
        write_access = (@defined($flags)
                        ? $flags & FAULT_FLAG_WRITE : $write_access)
        address =  $address
}
```

# Example of a tapset function

- without syscall tapset:

```
probe kernel.function("handle_mm_fault") {
  numFaults[probefunc()]++ }
```

- using syscall tapset:

```
probe vm.pagefault {
  numFaults[name]++ }
```

# Tapset examples

- syscall.*
  - Probes each system call, provides name and argstr
- vm.*
  - Used to probe memory-related events
- socket.*
  - Probes socket-related events

# Example: syscall tapset cont.

- For every system call, syscalls.stp provides:
  - name: syscall name
  - argstr: argument values encoded in a string
  - individual arg values
  - Retstr: ( for return ) return value encoded in a string

```
probe syscall.* {
   printf("%s(%s)\n", name, argstr)
}
probe syscall.*.return {
   printf(“%s returns %s\n”, name, retstr)
}
```

# Example: syscall tapset cont.

```
probe syscall.read =kernel.function("sys_read")
{
        name = "read"
        fd = $fd
        buf_uaddr = $buf
        count = $count
        argstr = sprintf("%d, %s, %d", $fd, …
```

- Probe in script:

```
probe syscall.read {
  rd_bytes_requested += count
}
```

# Tapsets

- tapset::iosched - systemtap IO scheduler probe points
- tapset::irq - Systemtap probes for IRQ, workqueue, etc
- tapset::kprocess - systemtap kernel process probe points
- tapset::netdev - systemtap network device probe points
- tapset::nfs - systemtap NFS client side probe points
- tapset::nfsd - systemtap NFS server side probe points
- tapset::pagefault - systemtap pagefault probe points
- tapset::perf - systemtap perf probe points
- tapset::rpc - systemtap SunRPC probe points
- tapset::scsi - systemtap scsi probe points
- tapset::signal - systemtap signal probe points
- tapset::snmp - Systemtap simple network management protocol probe points
- tapset::tcp - systemtap tcp probe points
- tapset::udp - systemtap udp probe points

# Embedded C code

- Embedded C is copied unchanged from your script to the module .c file.
- Embedded C is allowed only in tapsets or in scripts compiled with stap -g (guru mode).
- Embedded C code is usually used inside a function that starts with %{ and ends with %}

# Example: Embedded C code

```
%{
#include <net/sock.h>
#include <net/tcp.h>
#include <net/ip.h>
#include <asm/byteorder.h>
%}

function sk_info:string(sock:long)
%{
        struct inet_sock *inet = (struct inet_sock *)((long)THIS->sock);
        unsigned char saddr[4], daddr[4];

        memcpy(saddr, &inet->saddr, sizeof(saddr));
        memcpy(daddr, &inet->daddr, sizeof(daddr));
        sprintf(THIS->__retvalue, "%d.%d.%d.%d:%d -> %d.%d.%d.%d:%d",
           saddr[0], saddr[1], saddr[2], saddr[3], ntohs(inet->sport),
           daddr[0], daddr[1], daddr[2], daddr[3], ntohs(inet->dport));
%}
```

# References

- SystemTap documentation
  - Tutorial
  - Beginner's Guide
  - Language Reference
  - Tapset Reference
    - http://sourceware.org/systemtap/documentation.html

- Redbook: SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems
  - http://www.redbooks.ibm.com/redpapers/pdfs/redp4469.pdf

# References Cont.

- IBM SystemTap Blueprints
  - http://publib.boulder.ibm.com/infocenter/lnxinfo/v3r0m0/topic/liaai/liaaiSystemTap.htm

- RHEL6 SystemTap Beginners Guide
- RHEL6 SystemTap Tapset Reference
  - http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/index.html

# References

There are man pages:

- stap
  - systemtap program usage, language summary
- stappaths
  - your systemtap installation paths
- stapfuncs
  - functions provided by tapsets
- stapprobes
  - probes / probe aliases provided by tapsets
- stapex
  - some example scripts