Systemtap times

July 2009

Frank Ch. Eigler  <fche@redhat.com>  
systemtap lead
why trace/probe

- to monitor future
- to debug present
- to analyze past
problem space

• specification: what data to gather

• execution: how to gather it
  – compiled? overheads? disruptiveness? portability?

• analysis: how to understand it
  – bulk trace? visualize? response? custom reporting?
in praise of generality I

• why programmable?
  – conditions & actions sometimes need to be:
    • expressive (“collect variable X, Y”; dereference complex pointer expression; format reports)
    • stateful (“elapsed time greater than recent average for this operation on that device”)
  – program artifact (script) easy to share, abstract
single idea

• what to watch for?
  – kernel.function("sys_open")
  – process("/bin/bash").begin
  – timer.s(10)

• what to do?
  – print something
  – remember something
  – change something
simple syntax

• probe EVENT { ACTION }

• actions are C/awk like, plus
  – $context variables
  – loops, conditions, functions
  – global variables (automatically locked)
  – escape to raw C for guru users

• stap foo.stp
in praise of generality II

• why unified?
  – some problem go beyond individual programs or subsystems
  – many kinds of event sources exist
    • kernel probes, timers, watchpoints, user-space probes, ...
    • each with its own API
  – events occur in many contexts
    • kernel responses to user-space occurrences
    • shared libraries used by many processes
examples

• http://sourceware.org/systemtap/examples/index.html
• http://sourceware.org/systemtap/wiki/WarStories

• ordinary
  - log events, filtered + correlated + summarized
  - call graphs with variable dumps
  - measure times/values, indexed by anything
  - graph cpu/net/disk utilization, act upon thresholds

• esoteric
  - kernel-enforced file naming policy filters
  - security bug band-aids
recent developments

- rich symbolic probing user-space programs
- attaching to user + kernel markers, tracepoints
- organizing more samples, documentation
- easing deployment: compile server, debuginfo-less operations
user-space probing

- finally, system-wide, seamless, symbolic
- based upon dwarf debugging data (gcc -g)
- dynamically instrument binaries, shared libraries, potentially at the statement level
- easily trace variables
- attach to sys/sdt.h dtrace markers too, as compiled into postgres, java, ...
user-space probing

• measure average dbms query execution times

```c
function time() { return gettimeofday_us() }
probe process("psql").function("SendQuery").call {
    entry[tid()] = time()
}
probe process("psql").function("SendQuery").return {
    tid = tid()
    if (! ([tid] in entry)) next
    query = user_string($query)
    queries[query] <<= time() - entry[tid]
    delete entry[tid]
}
/* and an “end” probe to format report */
user-space probing

probe end,error,timer.s(5) {
    printf("%2s %6s %-40s\n",
           "#", "uS", "query");
    foreach ([q] in queries- limit 10)
        printf("%2d %6d %-40s\n",
                @count(queries[q]),
                @avg(queries[q]), q)
    printf("\n");
    delete queries
}
<table>
<thead>
<tr>
<th>#</th>
<th>uS query</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>DELETE FROM num_result;</td>
</tr>
<tr>
<td>6</td>
<td>COMMIT TRANSACTION;</td>
</tr>
<tr>
<td>6</td>
<td>BEGIN TRANSACTION;</td>
</tr>
<tr>
<td>6</td>
<td>SELECT date '1999-01-08';</td>
</tr>
<tr>
<td>4</td>
<td>insert into toasttest values(decode(repeat('1234567890',10000),'escape'));</td>
</tr>
<tr>
<td>4</td>
<td>insert into toasttest values(repeat('1234567890',10000));</td>
</tr>
<tr>
<td>4</td>
<td>SELECT '' AS five, * FROM FLOAT8_TBL;</td>
</tr>
<tr>
<td>3</td>
<td>END;</td>
</tr>
<tr>
<td>3</td>
<td>BEGIN;</td>
</tr>
<tr>
<td>3</td>
<td>INSERT INTO TIMESTAMPTZ_TBL VALUES ('now');</td>
</tr>
</tbody>
</table>
operation part 1

- compile probe script foo.stp:
  - parse script
  - combine it with tapset (library of scripts by experts)
  - elaborate it with debugging information, probe catalogues, event source metadata
  - generate C code with safety checks
  - compile into kernel module with kbuild
  - result: vanilla kernel module
operation part 2

- run probe module foo.ko:
  - load into kernel
  - detach (flight-recorder mode) or consume trace live
  - unload

- probe module may be cached, reused, shared with other machines running same kernel

- sysadmins can authorize others to run precompiled modules
under construction

- system-wide backtracing for deep profiling
- java probing & backtracing
- unprivileged user support
- gui-controlled monitoring
- better quality and smaller quantity of debuginfo
- interface to other kernel event sources: perfctr, ftrace
- non-kernel-ko backends for simple scripts
samples/documentation

- samples installed, categorized, also online
  - http://sourceware.org/systemtap/examples/index.html
- “beginner's guide”
  - http://tinyurl.com/ar8wat
- wiki
  - http://sourceware.org/systemtap/wiki