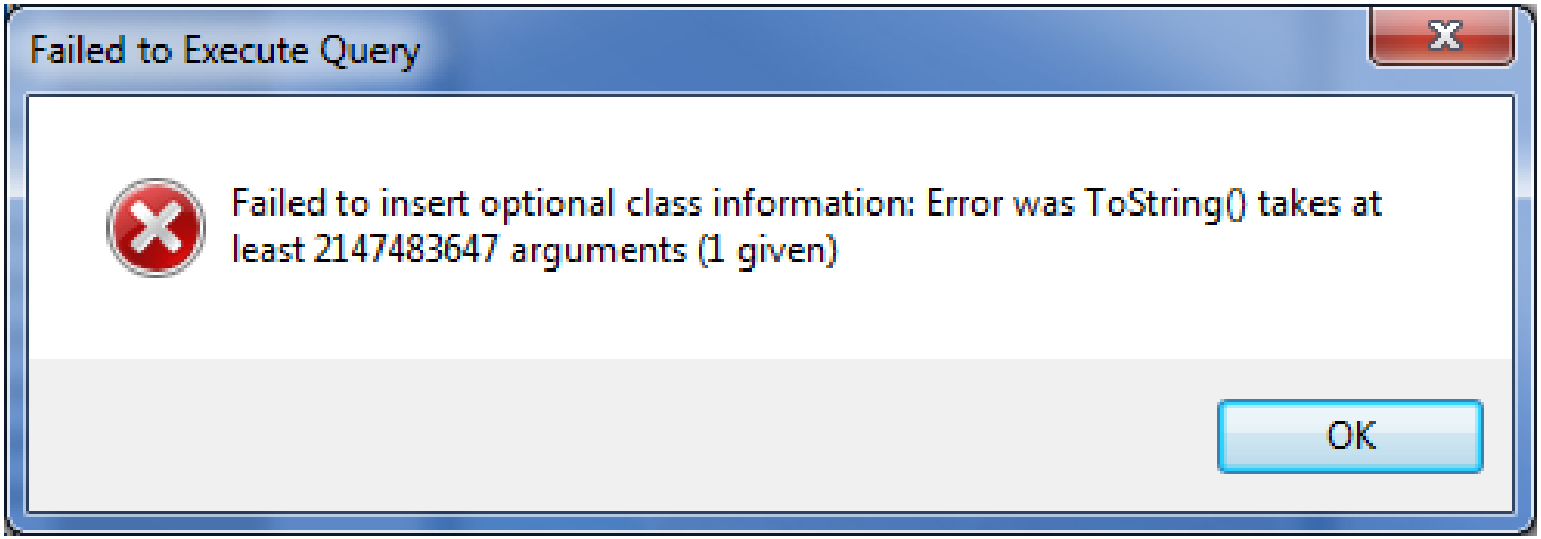
Probing PostgreSQL with SystemTap and DTrace

Joshua Tolley – eggyknap – End Point Corporation

May 21, 2010

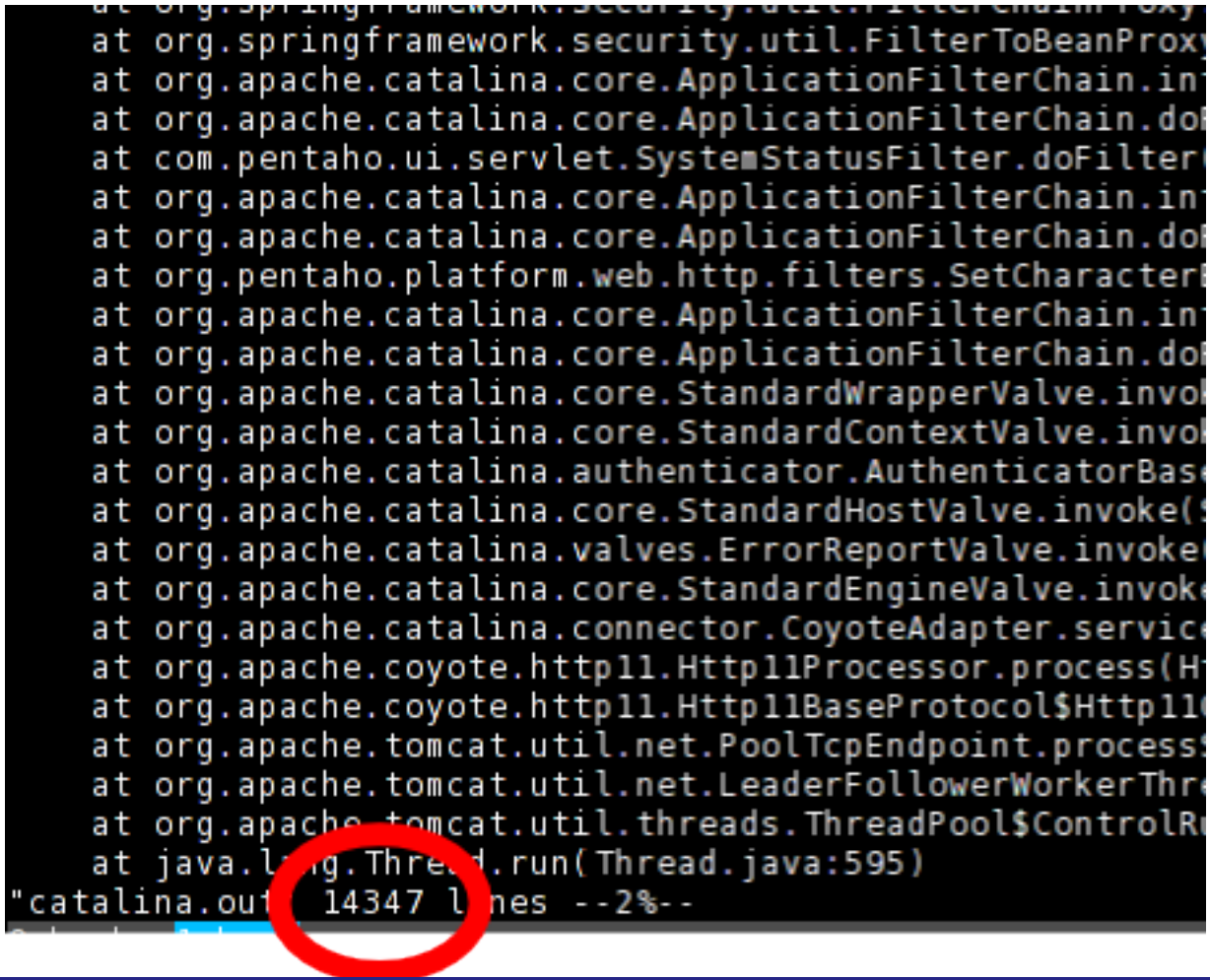
The problem

Sometimes error messages are inadequate



The problem

Sometimes error logs aren't much better



Typical solutions

Often there exists a tool to get the information you're after, but...

Typical solutions

Often there exists a tool to get the information you're after, but...

- You've never heard of it

Typical solutions

Often there exists a tool to get the information you're after, but...

- You've never heard of it
- No one else has ever heard of it

Typical solutions

Often there exists a tool to get the information you're after, but...

- You've never heard of it
- No one else has ever heard of it
- Even when people *have* heard of it, its options, usage, and output differ completely from the other tools you've used to get to this point, so integration is painful

Typical solutions

Often there exists a tool to get the information you're after, but...

- You've never heard of it
- No one else has ever heard of it
- Even when people *have* heard of it, its options, usage, and output differ completely from the other tools you've used to get to this point, so integration is painful
- The more specific the tool, the more output you need to sort through

Enter Dynamic Tracing...

The basics

Dynamic tracing allows users to instrument production systems in (ideally) whatever ways they want, without (ideally) breaking things in the process

The basics

Important packages

The basics

Important packages

- DTrace
  - Available on [Open]Solaris, FreeBSD, and OSX
  - Solaris' version is mature and widely used
  - FreeBSD's version is less mature than Solaris, and thus has significant limitations
  - Who runs OSX servers, anyway?
  - Any guesses how much life OpenSolaris has left?

The basics

Important packages

- DTrace
  - Available on [Open]Solaris, FreeBSD, and OSX
  - Solaris' version is mature and widely used
  - FreeBSD's version is less mature than Solaris, and thus has significant limitations
  - Who runs OSX servers, anyway?
  - Any guesses how much life OpenSolaris has left?
- SystemTap
  - Linux only
  - Less mature than DTrace
  - Requires kernel tracing patches and/or debug information to be really useful

The basics

Important packages

- DTrace
  - Available on [Open]Solaris, FreeBSD, and OSX
  - Solaris' version is mature and widely used
  - FreeBSD's version is less mature than Solaris, and thus has significant limitations
  - Who runs OSX servers, anyway?
  - Any guesses how much life OpenSolaris has left?
- SystemTap
  - Linux only
  - Less mature than DTrace
  - Requires kernel tracing patches and/or debug information to be really useful
- There are others (ProbeVue on AIX, other Linux packages with the same goal)

The basics

- Software (kernel, libraries, PostgreSQL, etc.) is equipped with various probe points
- Simple, C-like scripting language includes variables, functions, screen output, control structures, etc.
- User-supplied script describes which probes are used and what to do with them
- Compiled and run at the kernel level
- Runtime environment includes security and fault tolerance protections
- Ideally monitoring overhead is minimal for things that are probed, and nothing for unused probes

Some differences

- DTrace compiles to bytecode run in a kernel-level interpreter, SystemTap to native code run as a kernel module
- DTrace is far more mature
- SystemTap requires utrace kernel patch (available by default in Fedora, RHEL, CentOS)
- SystemTap can put probes anywhere you want, provided you have debug info. DTrace is limited to defined probes and a limited set of other places (function entry/exit, for instance)
- DTrace funs on several operating systems, and is CDDL-licensed. SystemTap is Linux only, and GPL

An example...

"Hello, World!", DTrace style

DTrace script

```
BEGIN
{
 trace("Hello, World!");
 exit(0);
}
```

Output

```
dtrace -s Hello.d
dtrace: script 'Hello.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0      1                      :BEGIN   Hello, World!
```

A more complex example and its output

```
syscall:::entry {
    @num[probefunc] = count();
}
```

```
  lwp_self                                          1
  write                                            33
  sigaction                                        33
  lwp_sigmask                                      53
  ioctl                                            95
```

Pieces of a DTrace script

- Probe declaration (provider:module:function:name). What probe are we looking for?

  `syscall::write:entry`

  `lockstat:::spin-acquire`

  `postgresql1234:::transaction-start`

Pieces of a DTrace script

- Probe declaration (provider:module:function:name). What probe are we looking for?

  ```
  syscall::write:entry
  lockstat:::spin-acquire
  postgresql1234:::transaction-start
  ```

- Predicate (optional). Are there only certain conditions where this probe is interesting?

  ```
  /execname == "postgres"/
  /self->fd = 1/
  ```

Pieces of a DTrace script

- Probe declaration (provider:module:function:name). What probe are we looking for?

  ```
  syscall::write:entry
  lockstat:::spin-acquire
  postgresql1234:::transaction-start
  ```

- Predicate (optional). Are there only certain conditions where this probe is interesting?

  ```
  /execname == "postgres"/
  /self->fd = 1/
  ```

- Code to run when probe fires

  ```
  printf("Hello, world!");
  ```

These scripts can get extremely complex.

- Thousands or millions of possible probes
- Speculative tracing
- Aggregates
- On-the-fly modification of processes' data

SystemTap

SystemTap looks somewhat similar:

SystemTap "Hello, World!"

```
probe begin
{
    printf("Hello, World!\n")
    exit()
}
```

Another example

```
probe vfs.read.return {
  if ($return>0) {
    if (devname!="N/A") {
      io_stat[pid(),execname(),uid(),ppid(),"R"]
       += $return
      device[pid(),execname(),uid(),ppid(),"R"]
       = devname
      read_bytes += $return
    }
  }
}
```

That example was part of a much longer script which:

- Captures each filesystem read and write, from each process
- Records the process name, device name, operation type, and size
- Every five seconds, prints average activity over the five second period, along with statistics for each process that used the disk in that time

SystemTap script components

SystemTap scripts declare a probe and follow it with code, just like DTrace (though SystemTap doesn't support DTrace-style predicates). Probe declarations come in many forms:

- Tapset references – aliases to other probes, defined in tapset scripts
  - `timer.ms(200)`
  - `ioblock.done`
- System call probes
  - `syscall.write.begin`
  - `syscall.*`

SystemTap script components

- Process probes using PID or path to executable
  - `process("/usr/bin/postgres").function("eqjoinsel")`
  - `process(31337).function("foo")`
- Marker-based probes (allows SystemTap to use defined DTrace probes, without code changes in the application)
  - `process(5783).mark("transaction__start")`

What can happen inside a probe function?

Probes can carry with them a set of parameters to describe the current system status.

SystemTap code

```
probe process(123).mark("query__parse__start")
{
    printf("query: %s\n",
        user_string($arg1));
}
```

Output

```
query: drop table if exists pgbench_branches
query: create table pgbench_branches(bid int ...
query: drop table if exists pgbench_tellers
query: create table pgbench_tellers(tid int ...
query: drop table if exists pgbench_accounts
query: create table pgbench_accounts(aid int ...
query: drop table if exists pgbench_history
query: create table pgbench_history(tid int...
query: begin
query: insert into pgbench_branches(bid,...
query: insert into pgbench_tellers(tid,bid,...
```

Debug info

In SystemTap, thanks to debug info (DWARF), it's possible to examine any *local* variable. Thus far I've been unable to read variables not local to the function I'm probing.

Example

```
probe process("/usr/bin/postgres").
    function("_bt_first")
{
    printf("scanning %d keys on index OID %d\n",
        $scan->numberOfKeys,
        $scan->indexRelation->rd_id);
}
```

Debug info

Output

```
scanning 1 keys on index OID 2679
scanning 3 keys on index OID 2696
scanning 3 keys on index OID 2696
scanning 3 keys on index OID 2696
scanning 3 keys on index OID 2696
scanning 2 keys on index OID 2663
```

PostgreSQL probe points

PostgreSQL defines several markers for probing:

- transaction begin, commit, and rollback
- query start/end, parsing start/end, rewriting start/end, planning start/end, executing start/end
- checkpoints start/end, and each phase of checkpointing (clog, subtransaction, multiXact, buffer, twophase)
- buffers: read, flush, write-dirty
- wal buffer start/end, xlog insert, xlog switch
- storage manager read/write
- sort start/end
- locking: lock and lwlock acquire, release, wait start/end, acquire failures, deadlock

PostgreSQL probe points

More possibilities arise when PostgreSQL probes are combined with kernel-level probes. Since SystemTap can gather data from any function, provided debug information, there are still more interesting places to look for data

PostgreSQL probe points

More possibilities arise when PostgreSQL probes are combined with kernel-level probes. Since SystemTap can gather data from any function, provided debug information, there are still more interesting places to look for data

- Kernel-level locking mechanisms
- Per-device fsync() timing
- Per-file, per-device, per-process IO statistics
  - Compare traffic levels across WAL, tablespaces, clog, stats file
- Buffer cache hit rates
  - Buffer read rates might conceivably give a good idea of system file cache hit rates

In short, this stuff is pretty neat

questions?