Debugging Node.js in Production:
Postmortem Debugging and Performance Analysis

Fluent 2012

David Pacheco (@dapsays)
Joyent

The Rise of Node.js

Joyent

- We see Node.js as the confluence of three ideas:
  - JavaScript's friendliness and rich support for asynchrony (i.e., closures)
  - High-performance JavaScript VMs (e.g., V8)
  - Time-tested system abstractions (i.e. Unix)
- Event-oriented model delivers consistent performance in the presence of long latency events (i.e. no artificial latency bubbles)
- Node.js is displacing C for a lot of high-performance, highly reliable **core infrastructure software** (at Joyent alone: DNS, DHCP, SNMP, LDAP, key value stores, public-facing web services, ...).
- This has been great for rapid development, but historically has come with a cost in **debuggability**.

The case of the run-away Node service

Joyent

- February, 2011: Joyent is preparing to launch no.de (free PaaS)
- Cloud Analytics service is intermittently unresponsive
- Traced the problem to a rogue data aggregator (one of 16) using 100% of 1 CPU core. Not responding using any means we had of querying it (HTTP, AMQP).
- How do you debug this?

Debugging a run-away Node program

Joyent

- What's the stack look like?

```
v8::internal::Runtime::SetObjectProperty+0x36d()
v8::internal::Runtime_SetProperty+0x73()
0xfe7601f6()
0xfbff31d8()
0xfc468f59()                          ...
0xfe8e51cf()                          ev_run+0x406()
0xfe760841()                          uv_run+0x1c()
0xfe8e3dc8()                          node::Start+0xa9()
0xfe8e24a4()                          main+0x1b()
   ...                                _start+0x83()
```

- What system calls is it making? (None (!))
- Can't add logging, because we don't know how to reproduce.
- ... but it's still exhibiting these symptoms. Can't we figure out why?!

Imagine a simpler problem

Joyent

- The software: a moderately complex concurrent service (where concurrent requests can affect one another).
- The deployment: in production with anywhere from a handful to a hundred instances.
- The problem: every day or so, one of the instances crashes, leaving behind just a stacktrace where an assertion was blown.
- How do you debug this (assuming the stacktrace is not sufficient)?

A first approach

Joyent

- Add instrumentation (printf) and redeploy.
- How easy is it to deploy a new version? How risky is it? What's the impact? Are you sure you'll only need to do this once?
- What if it's a very common code path that you need to instrument?
- What if this isn't your code, but a customer's that you're supporting? You have no control over deployment, and you lose credibility each time you ask a customer to do this.
- But if you're lucky and the problem is relatively simple, this can work okay.

A better approach

Joyent

- For C programs, we have rich tools for *postmortem* analysis.
- When a program crashes, the OS saves a **core dump**. The program can be immediately restarted to **restore service quickly** so that engineers can **debug the problem asynchronously**.
- Using the debugger on the core dump, you can inspect **all** internal program state: global variables, threads, and objects.
- Historically, existing tools have been unable to meaningfully observe JIT'd environments like Node, and those environments have not developed equally rich tools to address these problems.
- Node is not alone! The state of the art is no better in Python or Ruby, and not nearly solved for Java or Erlang either.

Why is it hard?

Joyent

- Native abstractions != JavaScript abstractions:
  Native postmortem debugging doesn't need to present any abstractions that don't already exist in the system (symbols and functions). A JavaScript VM would need to present the native structures as their JavaScript counterparts.
- Some of these abstractions don't even exist explicitly in the language itself (like JavaScript's queue of pending events).
- You either need a custom (or extensible) debugger that can iterate VM internal structures from a core file, or the VM itself needs to serialize JavaScript state (heap, stack, etc.) when the program crashes (and on-demand) and provide a tool to examine that.

Postmortem debugging for Node.js

Joyent

- mdb_v8: based on MDB, the illumos modular debugger.
- Given a core file (from gcore(1), a segfault, abort(3C)), examine:
  - Current call stack, including JavaScript functions and arguments.
  - Given a pointer, print out as a C++ object, or print its JavaScript counterpart.
  - Scan the heap to identify how many instances of each object type exist. (incredible visibility into memory usage)
- Implementation:
  - V8 (libv8.a) includes a small amount (a few KB) of metadata that describes the heap's classes, type information, and class layouts.
  - mdb_v8 knows how to identify stack frames, iterate function arguments, iterate object properties, and walk basic V8 structures (arrays, functions, strings).
  - mdb_v8 uses the debug metadata encoded in the binary to avoid hardcoding the way heap structures are laid out in memory.
- Demo

Remember that run-away Node program?

Joyent

- When we first saw this in February, 2011, we had no way to peer inside and see what this program was doing. We saved a core dump, in case we might one day have the technology to read it. We also added other instrumentation, and we expected to see it again shortly (since we saw it so quickly in the first place).
- We didn't see it again until October, 2011, while the mdb_v8 work was underway. We applied what we had to a new core file.

And the winner is:

Joyent

```
> ::jsstack
8046a9c <anonymous> (as exports.bucketize) at lib/heatmap.js position 7838
8046af8 caAggrValueHeatmapImage at lib/ca/ca-agg.js position 48960
...
> 8046a9c::jsframe -v
8046a9c <anonymous> (as exports.bucketize)
    func: fc435fcd
    file: lib/heatmap.js
    posn: position 7838
    arg1: fc070719 (JSObject)
    arg2: fc070709 (JSArray)
> fc070719::jsprint
{
    base: 1320886447,
    height: 281,
    width: 624,
    max: 11538462,
    min: 11538462,
    ...
}
```

Invalid input resulted in infinite loop in JavaScript
Time to root cause: 10 minutes

Debugging live programs

Joyent

- The same postmortem technology can be used to inspect live state without disrupting the running program using gcore(1).
- Sometimes, you want something to trace runtime *activity*.

DTrace

Joyent

- Provides comprehensive tracing of both kernel and application-level events in **real-time**.
- Scales arbitrarily with the number of traced events.
  (first class *in situ* data aggregation)
- Suitable for production systems because it's **safe**, has minimal overhead (usually no disabled probe effect), and can be enabled/disabled dynamically (no application restart required).
- Open-sourced in 2005. Available on illumos (and Solaris-derived systems), BSD, and MacOS (Linux ports in progress).

DTrace example: MySQL query latency

Joyent

```
# dtrace –n ‘
mysql*:::query-start { self->start = timestamp; }
mysql*:::query-done /self->start/ {
 @[“nanoseconds”] = quantize(timestamp – self->start);
 self->start = 0;
}’
```

```
nanoseconds
           value  ------------- Distribution ------------- count
            1024 |                                         0
            2048 |                                         16
            4096 |@                                        93
            8192 |                                         19
           16384 |@@@                                      232
           32768 |@@                                       172
           65536 |@@@@@@                                   532
          131072 |@@@@@@@@@@@@@@@@@@@                      1513
          262144 |@@@@@                                    428
          524288 |@@@                                      258
         1048576 |@                                        127
         2097152 |@                                        47
         4194304 |                                         20
         8388608 |                                         33
        16777216 |                                         9
        33554432 |                                         0
```

Profiling Node with DTrace

Joyent

- DTrace provides a “ustack()” action for collecting a user-level stacktrace at the given probe point.
- This is useful when debugging to see who’s doing what (e.g., where am I calling malloc(3C), or who’s calling open(2)).
- Can also be used for profiling using “profile” provider that fires a probe N times per second on each CPU.
- What about JIT’d code?

ustack helpers

Joyent

- For JIT'd code, DTrace supports **ustack helper** mechanism, by which the VM itself includes logic to translate from

  (frame pointer, instruction pointer) -> human-readable function name

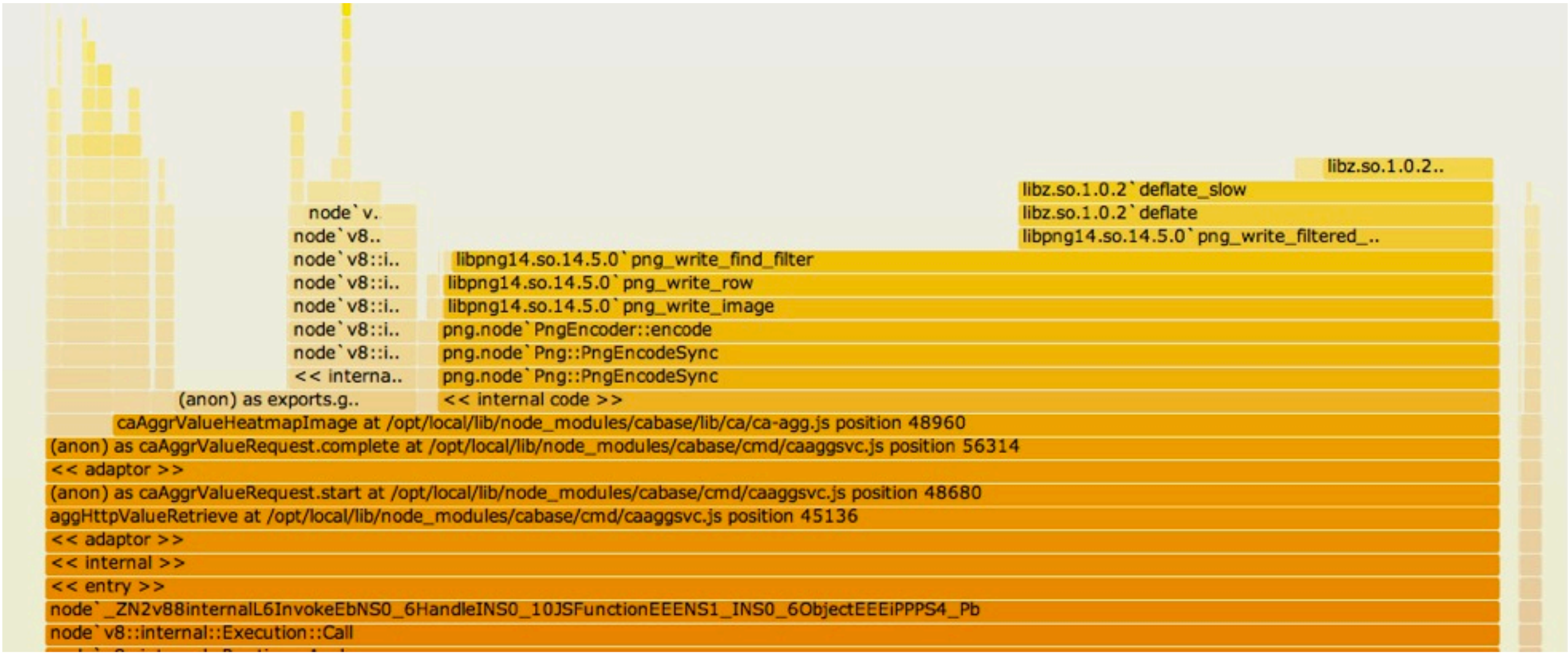
- When jstack() action is processed in probe context (in the kernel), DTrace invokes the helper to translate frames:

Before	After
0xfe772a8c	toJSON at native date.js position 39314
0xfe84d962	BasicJSONSerialize at native json.js position 8444
0xfea6b6ed	BasicSerializeObject at native json.js position 7622
0xfe84db11	BasicJSONSerialize at native json.js position 8444
0xfeaba5ee	stringify at native json.js position 10128

Node.js Flame Graph

Joyent

- Visualizing profiling output:



- http://www.cs.brown.edu/~dap/ca-flamegraph.svg

V8 ustack helper

Joyent

- The ustack helper has to do much of the same work that mdb_v8 does to identify stack frames and pick apart heap objects.
- The implementation is written in D, and subject to all the same constraints as other DTrace scripts (and then some): no functions, no iteration, no if/else.
- Particularly nasty pieces include expanding ConsStrings and binary searching to compute line numbers.
- The helper only depends on V8, not Node.js. With MacOS support for ustack helpers, we could use the same helper to profile webapps running under Chrome!

More real-world examples

Joyent

- The infinite loop problem we saw earlier was debugged with mdb_v8, and could have also been debugged with DTrace.
- @izs used mdb_v8‘s heap scanning to zero in on a memory leak in Node.js that was seriously impacting several users, including Voxer.
- @mranney (Voxer) has used Node profiling + flame graphs to identify several performance issues (unoptimized OpenSSL implementation, poor memory allocation behavior).
- Debugging RangeError (stack overflow, with no stack trace).

Final thoughts

Joyent

- Node is a great environment for building complex system software and distributed systems. But in order to achieve the reliability we expect from such systems, **we must be able to understand both fatal and non-fatal failure in production** from the first occurrence.
- One year ago: we had no way to solve the “infinite loop” problem without adding more logging and hoping to see it again.
- Now, we have tools to inspect both running and crashed Node programs (mdb_v8 and the DTrace ustack helper), and we’ve used them to debug problems in minutes that we either couldn’t solve at all before or which took days or weeks to solve.
- Future work: there’s lots more heap analysis to do, including finding variables by name (for globals and module-”globals”)

- Thanks:
  - @bcantrill for ::findjsobjects
  - @mraleph for help with V8 and landing patches
  - @izs and the Node core team for help integrating DTrace and MDB support
  - @mranney and Voxer for pushing Node hard, running into lots of issues, and helping us refine the tools to debug them. (God bless the early adopters!)

- For more info:
  - http://dtrace.org/blogs/dap/2012/04/25/profiling-node-js/
  - http://dtrace.org/blogs/dap/2012/01/13/playing-with-nodev8-postmortem-debugging/
  - https://github.com/joyent/illumos-joyent/blob/master/usr/src/cmd/mdb/common/modules/v8/mdb_v8.c
  - https://github.com/joyent/node/blob/master/src/v8ustack.d

Debugging Node.js in Production:
Postmortem Debugging and Performance Analysis

Fluent 2012

David Pacheco (@dapsays)
Joyent