Joyent

Node.js in Production:
Postmortem Debugging and Performance Analysis

OSCON 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, in the form of streams)
- 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 highly reliable, high performance **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

- Check the logs?
- Check syscall activity (truss/strace)?

Debugging a run-away Node program

Joyent

- Check thread stacks:

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

Debugging a run-away Node program

Joyent

- Can add more logging, but we don’t know how to reproduce it.
- ... 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 (that is, where concurrent requests can affect one another).
- The deployment: in production, 10s to 100s of instances.
- The problem: ~once/day, one of the instances crashes, leaving behind a stacktrace where an assertion was blown.
- Assuming the stacktrace and existing logs are not enough, how do you debug this?

A first approach

Joyent

- Add instrumentation (console.log) 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 don't control deployment, and you lose credibility each time you ask a customer to try again.)
- If you're lucky or if 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.
- Can also use the same tools with a live process.
- Can’t we do this with Node.js?

Debugging dynamic environments

Joyent

- Historically, native postmortem tools have been unable to meaningfully observe dynamic environments like Node.
- Few dynamic environments have developed rich tools to address these problems for their own domains.
- Node is not alone! The state of the art is no better in Python, Ruby, or PHP, and not nearly solved for Java or Erlang either.

Why is debugging hard for JIT'd code?

Joyent

- Need to translate native abstractions (symbols, functions, structs) into JavaScript counterparts (variables, Functions, Objects)
- Some abstractions don't even exist explicitly in the language itself. (e.g., JavaScript's event queue)

mdb_v8: postmortem debugging for Node

Joyent

- Based on MDB, the illumos modular debugger.
- Prints call stacks, including native C++ **and** JavaScript functions and arguments.
- Given a pointer, prints out as a C++ object **and** its JS counterpart.
- Scans the heap to identify how many instances of each object type exist (incredible visibility into memory usage).
- Demo

Remember that run-away Node program?

Joyent

- In February, 2011, we had essentially no way to see what this program was doing.
- We saved a core dump in case we might one day have a way to read it. We also added instrumentation in case we saw it again. (We expected to see it again very soon after going to production.)
- We didn’t see it again until October, while the mdb_v8 work was underway. So 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

- Postmortem tools can be applied to live processes, and core files can be generated for running processes.
- Examining processes and core dumps is useful for many kinds of failure, but sometimes you want to trace runtime *activity*.

DTrace

Joyent

- Provides comprehensive tracing of kernel and application-level events in **real-time** (from “thread on-CPU” to “Node GC done”)
- 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: Node GC time, per GC

```
# dtrace –n ‘
node*:::gc-start { self->start = timestamp; }
node*:::gc-done/self->start/{
 @[“microseconds”] = quantize((timestamp – self->start) / 1000);
 self->start = 0;
}’
```

```
microseconds
           value  ------------- Distribution ------------- count
              32 |                                         0
              64 |@@@@@                                    19
             128 |@@                                       6
             256 |@@                                       6
             512 |@@@@                                     13
            1024 |@@@@@                                    17
            2048 |@@@@@@@                                  24
            4096 |@@@@@@@@                                 29
            8192 |@@@@@                                    16
           16384 |@                                        5
           32768 |@                                        3
           65536 |                                         1
          131072 |@                                        3
          262144 |                                         0
```

Profiling Node with DTrace

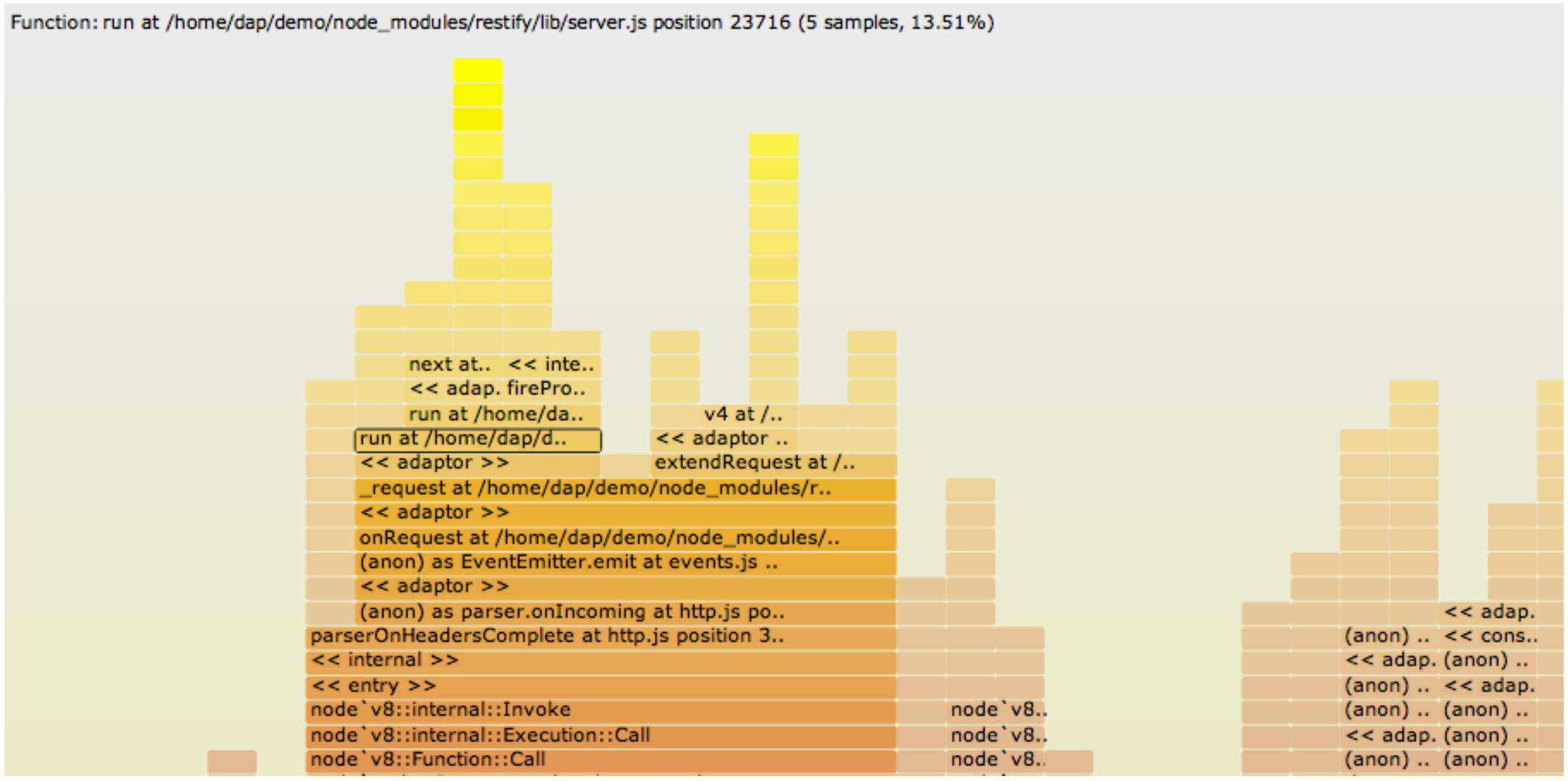
Joyent

- “profile” provider: probe that fires N times per second per CPU
- `ustack()`/`jstack()` actions: collect user-level stacktrace when a probe fires.
- Low-overhead runtime profiling (via stack sampling) that can be turned on and off without restarting your program.
- Demo.

Node.js Flame Graph

Joyent

- Visualizing profiling output:



- Full, interactive version: http://www.cs.brown.edu/~dap/restify-flamegraph.svg

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 0.7 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 for rapidly building complex or distributed system software. 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.
- But the postmortem tools are still primitive (like a flashlight in a dark room). Need better support from the VM.

- 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

Joyent

Node.js in Production:
Postmortem Debugging and Performance Analysis

OSCON 2012

David Pacheco (@dapsays)
Joyent

mdb_v8 implementation

Joyent

- V8 (libv8.a) includes a small amount (a few KB) of metadata that describes the heap's classes, type information, and class layouts. (Small enough to include in all builds, including **production**.)
- 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. (Still has intimate knowledge of things like property iteration.)

More on DTrace stack traces

Joyent

- `ustack()`: DTrace looks at (%ebp, %eip) and follows frame pointers to the top of the stack (standard approach). Asynchronously, looks for symbols in the process's address space to map instruction offsets to function names:

  `0x80ed9ab` becomes `malloc+0x16`

- Great for C, C++. Doesn't work for JIT'd environments.
  - Functions are compiled at runtime => they have no corresponding symbols
    => the VM must be called upon at runtime to map frames to function names
  - Garbage collection => functions themselves move around at arbitrary points
    => mapping of frames to function names must be done "synchronously"

- `jstack()`: Like ustack(), but invokes VM-specific **ustack helper**, expressed in D and attached to the VM binary, to resolve names.

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

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 from profile probes, we could use the same helper to profile webapps running under Chrome!)