goto; conference aarhus

Dynamic Languages In Production: Progress And Open Challenges

Bryan Cantrill (@bcantrill)
David Pacheco (@dapsays)

Joyent

INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE

gotocon.com

Dynamic languages: In the beginning...

Joyent

Dynamic languages: In the beginning...

Joyent

John McCarthy, 1927 - 2011

Dynamic languages: In the beginning...

Joyent

"The existence of an interpreter and the absence of declarations makes it particular natural to use LISP in a time-sharing environment. It is convenient to define functions, test them, and re-edit them without ever leaving the LISP interpreter."

— John McCarthy, 1927 - 2011

Dynamic languages

Joyent

- From their inception, dynamic and interpreted languages have enabled higher programmer productivity
- ...but for many years, limited computing speed and memory capacity confined the real-world scope of these languages
- By the 1990s, with faster microprocessors, better DRAM density and improved understanding of virtual machine implementation, the world was ready for a breakout dynamic language...
- Java, introduced in 1995, quickly became one of the world's most popular languages — and in the nearly two decades since Java, dynamic languages more generally have blossomed
- Dynamic languages have indisputable power…
- ...but their power has a darker side

Before the beginning

Joyent

Before the beginning

Joyent

Sir Maurice Wilkes, 1913 - 2010

Before the beginning

Joyent

“As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.”

—Sir Maurice Wilkes, 1913 - 2010

Dynamic languages

Joyent

- Debugging infrastructure for a dynamic language requires a great deal of VM and language specificity
- Such as this infrastructure has been created for dynamic languages, it has been aimed at the *developer* in *development*
- The *production* environment is much more constrained:
  - Deployed apps cannot be modified merely to debug them
  - Failures cannot be assumed to be reproducible
  - Failure modes themselves may be non-fatal or transient
- Given its constraints, how is production software *ever* debugged?

“The postmortem technique”

Joyent

“Experience with the EDSAC has shown that although a high proportion of mistakes can be removed by preliminary checking, there frequently remain mistakes which could only have been detected in the early stages by prolonged and laborious study. Some attention, therefore, has been given to the problem of **dealing with mistakes after the programme has been tried and found to fail**.”

—Stanley Gill, 1926 - 1975
“The diagnosis of mistakes in programmes on the EDSAC”, **1951**

Postmortem debugging

Joyent

- For native programs, we have rich tools for *postmortem* analysis of a system based on a snapshot of its state.
- This technique is so old (viz. EDSAC), the term for this state snapshot dates from the dawn of computing: it's a **core dump**.
- Once a core dump has been generated, either automatically after a crash or on-demand using gcore(1), 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 we do this with dynamic environments?

Debugging dynamic environments

Joyent

- Historically, native postmortem tools have been unable to meaningfully observe dynamic environments
- Such tools would need to translate native abstractions from the dump (symbols, functions, structs) into their higher-level counterparts in the dynamic environment (variables, Functions, Objects).
- Some abstractions don't even exist explicitly in the language itself. (e.g., JavaScript's event queue)
- These tools must *not* assume a running VM — they must be able to discern higher-level structure from *only* a snapshot of memory!
- While the postmortem problem remains unsolved for essentially all dynamic environments, we were particularly motivated to solve it for Node.js

The Rise of Node.js

Joyent

- We have found Node.js to displace 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, ...).
- Node.js is a good fit for these services because it represents 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)
- In developing and deploying our own software, we found lack of production debuggability to be Node's most serious shortcoming!

Aside: MDB

Joyent

- illumos-based systems like SmartOS and OmniOS have **MDB**, the modular debugger built specifically for postmortem analysis
- MDB was originally built for postmortem analysis of the operating system kernel and later extended to applications
- Plug-ins (“dmods”) can easily build on one another to deliver powerful postmortem analysis tools, e.g.:
  - **`::stacks`** coalesces threads based on stack trace, with optional filtering by module, caller, etc
  - **`::findleaks`** performs postmortem garbage collection on a core dump to find memory leaks in native code
  - **`::typegraph`** performs postmortem object type identification for native code by propagating type inference through the object graph
- Could we build a dmod for Node?

mdb_v8: postmortem debugging for Node

Joyent

- With some excruciating pain and some ugly layer violations, we were able to build **mdb_v8**
- With **`::jsstack`**, prints call stacks, including native C++ **and** JavaScript functions and arguments.
- With **`::jsprint`**, given a pointer, prints out as a C++ object **and** its JavaScript counterpart.
- With **`::v8function`**, given a JSFunction pointer, show the assembly for that function.

Heap profiling with mdb_v8

Joyent

- **`::findjsobjects`** scans the heap and prints a count of objects broken down by signature (i.e., property names).

```
OBJECT   #OBJECTS #PROPS CONSTRUCTOR: PROPS
fc3edc41         1      7 AMQPParser: frameBuffer, ...
fc42e4d5        91      4 Object: methodIndex, fields, ...
...
```

  Gives a coarse summary of memory usage, mainly used to spot red flags (e.g., many more of some object than expected).

- ***`ptr`*`::findjsobjects`** finds all objects similar to *`ptr`* (i.e., having the same properties as *`ptr`*). Used once a red flag is spotted to examine suspicious objects in more detail.

- **`::findjsobjects -p` *`propname`*** finds all objects with property *`propname`*. Very useful for finding and inspecting program state!

mdb_v8: How the sausage is made

Joyent

- mdb_v8 knows how to identify stack frames, iterate function arguments, iterate object properties, and walk basic V8 structures (arrays, functions, strings).
- 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 **production** builds.)
- 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.)

What did you say was in this sausage?

Joyent

- Goal: debugger module shouldn't hardcode structure offsets and other constants, but rather rely on metadata included in the "node" binary.
- Generally speaking, these offsets are computed at compile-time and used in inline functions defined by macros. So they get compiled out and are not available at runtime.
- The build process was modified to:
  - Generate a new C++ source file with references to the constants that we need, using extern "C" constants that our debugger module can look for and read.
  - Build this with the rest of libv8_base.a.
- Result: this "debug metadata" is embedded in $PREFIX/bin/node, and the debugger can read it directly from the core file.
- (Should) generally work for 32-bit/64-bit, different architectures, and no matter how complex the expressions for the constants are.

Problems with this approach

Joyent

- We strongly believe in the general approach of having the debugger grok program state from a snapshot, because it's **comprehensive** and has **zero runtime overhead**, meaning it works in production. (This is a constraint.)
- With the current implementation, the debugger module is built and delivered separately from the VM, which means that changes in the VM can (and do) break the debugger module.
- Additionally, each debugger feature requires reverse engineering and reimplementing some piece of the VM.
- Ideally, the VM would embed programmatic logic for decoding the in-memory state (e.g., iterating objects, iterating object properties, walking the stack, and so on) — without relying on the VM itself to be running.

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-derived systems like SmartOS and OmniOS, Solaris-derived systems, BSD, and MacOS (Linux ports in progress).

DTrace in dynamic environments

Joyent

- DTrace instruments the system *holistically*, which is to say, from the kernel, which poses a challenge for interpreted environments
- User-level statically defined tracing (USDT) providers describe semantically relevant points of instrumentation
- Some interpreted environments (e.g., Ruby, Python, PHP, Erlang) have added USDT providers that instrument the interpreter itself
- This approach is very fine-grained (e.g., every function call) and doesn't work in JIT'd environments
- We decided to take a different tack for Node.js...

DTrace for Node.js

Joyent

- Given the nature of the paths that we wanted to instrument, we introduced a function into JavaScript that Node can call to get into USDT-instrumented C++
- Introduces disabled probe effect: calling from JavaScript into C++ costs even when probes are not enabled
- Use USDT is-enabled probes to minimize disabled probe effect once in C++
- If (and only if) the probe is enabled, prepare a structure for the kernel that allows for translation into a structure that is familiar to node programmers

DTrace example: Node GC time, per GC

Joyent

```
# 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
```

DTrace probes in Node.js modules

Joyent

- Our technique is adequate for DTrace probes in the Node.js core, but it's very cumbersome for pure Node.js modules
- Fortunately, Chris Andrews has generalized this technique in his node-dtrace-provider module:

  https://github.com/chrisa/node-dtrace-provider

- This module allows one to declare and fire one's own probes entirely in JavaScript
- Used extensively by Joyent's Mark Cavage in his node-restify and ldap.js modules, especially to allow for measurement of latency

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.

DTrace 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 same debug metadata that's used for mdb_v8 is used for the helper, but unlike mdb_v8, the helper is embedded directly into the node binary (good!).
- 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!)

Profiling Node with DTrace

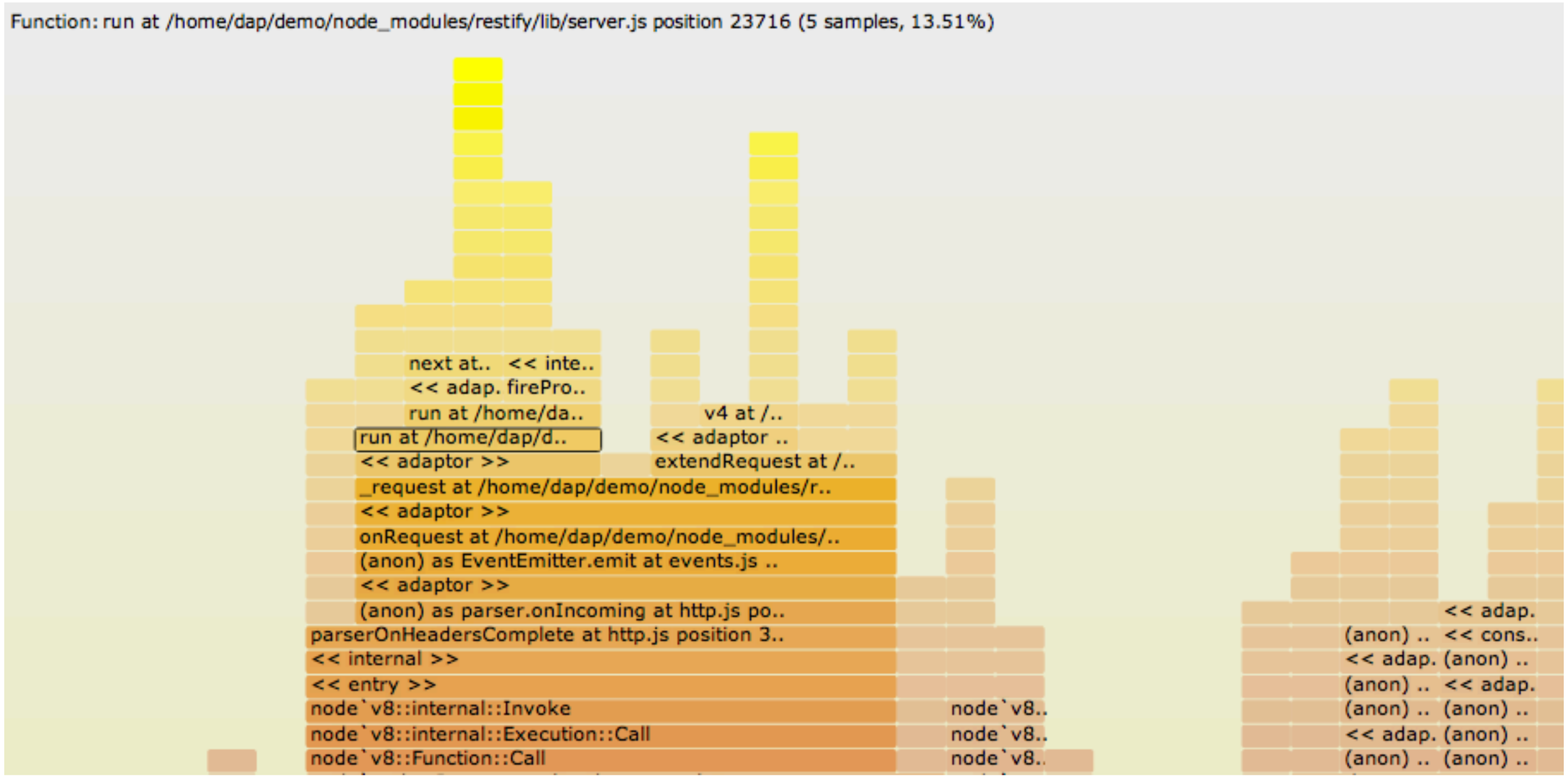
Joyent

- “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://bit.ly/NMQT1B

More real-world examples

Joyent

- A non-reproducible infinite loop we saw in production 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.
- We now 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.
- Somewhat unexpectedly, the postmortem approach has also showed promise on the pressing issue of production heap profiling
- But the postmortem tools are still primitive (like a flashlight in a dark room) and brittle; can we have the VM better encode or encapsulate the logic required to debug the programs it runs?

- Thanks:
  - @mraleph and @erikcorry for help with V8 and landing patches
  - @izs and the Node core team for help integrating DTrace and MDB support
  - @brendangregg for flame graphs
  - @chrisandrews for node-dtrace-provider
  - @mcavage for putting it to such great use in node-restify and ldap.js
  - @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