How Joyent Operates Node.js in Production

David Pacheco (@dapsays)
How Joyent Observes Node.js in Production

David Pacheco (@dapsays)
Node in production at Joyent

- Joyent Public Cloud (runs Smart Data Center)
- Smart Data Center (SDC)
- Manta
First principles

• When a problem is seen in production:
  • Restore service **immediately**
  • Root-cause the problem **fully** -- the **first** time
Easier said than done

• Production is far more constrained than development:
  • Cannot edit code and restart
    (hard to manage, and many problems are transient)
  • Cannot stop and attach a debugger
    (way too disruptive to service)
• Traditional debuggers (e.g., gdb) generally don’t work on dynamic environments
Kang

- Super simple toolset for browsing state of a running distributed system
- Used as the basis for dashboards
• “async”-like library
  • forEach, forEachParallel, pipeline, queue, barrier
• Stores state in an object (not closures)
• Can view state in kang, MDB, REPL, etc.
Example: Cloud Analytics Launch
Example: Cloud Analytics Launch

• Symptom: UI freeze
• Quick health check: 1 of 16 data aggregators not responding to application-level ping
• Process is 100% on-CPU, in userland
• Doing 0 syscalls (no network activity, no file activity)
• WTF do we do?
Core files

- Includes **all** of your Node program’s state
- Create with `gcore(1)`
- View with MDB
Core files for crashes

- Best of both goals (service restoration, debuggability)
- Configure Node to dump core on crashes:
  - v0.10.7 and earlier:
    On `uncaughtException`, call `process.abort()`
  - v0.10.8 and later:
    Use `--abort-on-uncaught-exception`
    (keeps stack intact)
Debugging core dumps

• Very different methodology than printf-debugging!
• Easy to get turned off because you don’t know where to start, but you’d be surprised what you theories you can prove (or disprove) with a core dump!
JavaScript heap analysis

- Core files afford more expensive analysis
- `findjsobjects` dumps a frequency count of all objects by "duck type"
- Good for finding big leaks
When static state isn’t enough

- Idea: add `console.log`, restart, reproduce
- Lots of problems with that, but the basic idea is good: get more information about what the program is doing
Bunyan

- Node.js logging library
- Format: Newline-separated JSON
- Bonus: runtime log snoopig with bunyan -p
DTrace basics

- System has hundreds of thousands of probes
- User writes script to take certain actions based on those probes
- Designed for production
  - Safe above all else
  - “Dynamic” => Zero overhead when disabled
  - *In situ* aggregation => low overhead when enabled
- Demo
Tracing Node.js

• Node DTrace provider has built-in probes:
  • http server request start/done
  • http client request start/done
  • garbage collection start/done
• See nhttpsnoop
### Example: tracing request latency

```bash
# /var/tmp/nhttpsnoop -cgsl
```

<table>
<thead>
<tr>
<th>TIME</th>
<th>PID</th>
<th>PROBE</th>
<th>LATENCY</th>
<th>METHOD</th>
<th>PATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ 0.068996]</td>
<td>15832</td>
<td>server -&gt;</td>
<td>-</td>
<td>GET</td>
<td>/jobs</td>
</tr>
<tr>
<td>[ 0.073913]</td>
<td>15832</td>
<td>server &lt;-</td>
<td>4.916ms</td>
<td>GET</td>
<td>/jobs</td>
</tr>
<tr>
<td>[ 0.396989]</td>
<td>16511</td>
<td>client -&gt;</td>
<td>-</td>
<td>GET</td>
<td>/configs/65879ef3</td>
</tr>
<tr>
<td>[ 0.397242]</td>
<td>29441</td>
<td>server -&gt;</td>
<td>-</td>
<td>GET</td>
<td>/configs/65879ef3</td>
</tr>
<tr>
<td>[ 0.409515]</td>
<td>29441</td>
<td>server &lt;-</td>
<td>12.272ms</td>
<td>GET</td>
<td>/configs/65879ef3</td>
</tr>
<tr>
<td>[ 0.409611]</td>
<td>16511</td>
<td>client &lt;-</td>
<td>12.622ms</td>
<td>GET</td>
<td>/configs/65879ef3</td>
</tr>
<tr>
<td>[ 0.411069]</td>
<td>16511</td>
<td>gc     &lt;-</td>
<td>0.863ms</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Tracing the system

• You can also use built-in probes!
  • memory allocation: malloc, sbrk, mmap
  • system activity: syscalls
  • process blocks

• Often, it’s useful to record a JavaScript stack trace when these events happen (or aggregate on the stacktrace)
Tracing your Node app

- You can add your own app-specific probes
Example: restify tracing

# ./restify-latency.d -p 25561
^C

ROUTE LATENCY (milliseconds)

<table>
<thead>
<tr>
<th>key</th>
<th>min</th>
<th>max</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>getconfigs</td>
<td>&lt; 0</td>
<td>:</td>
<td>6</td>
</tr>
<tr>
<td>headagentprobes</td>
<td>&lt; 0</td>
<td>:</td>
<td>5</td>
</tr>
<tr>
<td>listvms</td>
<td>&lt; 0</td>
<td>:</td>
<td>5</td>
</tr>
</tbody>
</table>

HANDLER LATENCY (milliseconds)

<table>
<thead>
<tr>
<th>key</th>
<th>min</th>
<th>max</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>listvms</td>
<td>&lt; 0</td>
<td>:</td>
<td>5</td>
</tr>
<tr>
<td>listvms addProxies</td>
<td></td>
<td>:</td>
<td>5</td>
</tr>
<tr>
<td>listvms bunyan</td>
<td></td>
<td>:</td>
<td>5</td>
</tr>
<tr>
<td>listvms checkMoray</td>
<td></td>
<td>:</td>
<td>5</td>
</tr>
<tr>
<td>listvms checkWfapi</td>
<td></td>
<td>:</td>
<td>5</td>
</tr>
<tr>
<td>listvms handler-0</td>
<td></td>
<td>:</td>
<td>5</td>
</tr>
<tr>
<td>listvms listVms</td>
<td></td>
<td>:</td>
<td>5</td>
</tr>
<tr>
<td>listvms loadVm</td>
<td></td>
<td>:</td>
<td>5</td>
</tr>
<tr>
<td>listvms parseAccept</td>
<td></td>
<td>:</td>
<td>5</td>
</tr>
<tr>
<td>listvms parseBody</td>
<td></td>
<td>:</td>
<td>5</td>
</tr>
<tr>
<td>listvms parseDate</td>
<td></td>
<td>:</td>
<td>5</td>
</tr>
<tr>
<td>listvms parseQueryString</td>
<td></td>
<td>:</td>
<td>5</td>
</tr>
<tr>
<td>listvms readBody</td>
<td></td>
<td>:</td>
<td>5</td>
</tr>
</tbody>
</table>
On-CPU performance

• We use DTrace-based profiling:

```bash
# dtrace -n profile-97/pid == $target/
{ @[jstack(80, 8192)] = count(); }'
```

• We visualize the results with **flame graphs** (demo)
Off-CPU performance

• Use DTrace to instrument start/done of asynchronous events (e.g., filesystem I/O, network request)

• Can visualize with a **heat map**
Development-time tips

• Compile *everything* with `-fno-omit-frame-pointer` (otherwise, *nothing* involving stacktraces works)

• Hang all state off a global singleton object (once you find that object, you can find all state)

• Store extra debugging state (e.g., `nretries`, `time_last_tried`)

• Use prefixes on object property names (helps `::findjsobjects` find specific objects -- and helps with `grep` too!)

• Use libraries that do these things (e.g., `vasync`)
Log analysis

- SDC and Manta logs are uploaded to Manta hourly
- We have some automated jobs, lots of ad-hoc jobs to analyze them
Platform dependencies

- We run **everything** on SmartOS (illumos-based).
- MDB: Nothing analogous on GNU/Linux (but TJ is working on reading Linux cores in MDB!)
- DTrace:
  - System probes, custom probes: illumos, OS X, BSD
  - JS stacks: illumos only
  - There’s SystemTap, prof, and the Oracle DTrace port... (unclear if any have JS support)
- bunyan, vasync, restify, kang: all work everywhere
Summary

• Fatal failures: core dumps
• Non-fatal failures:
  • Kang, core dumps
  • Logs: bunyan, bunyan -p
  • DTrace (system probes, Node probes, app probes)
• Performance (on-CPU, off-CPU)
• Memory analysis (both JS and native)
Summary: key tools and modules

• Tools:
  • **mdb**: modular debugger
  • **gcore**: generate core file for a process
  • **jsontool**: JSON from the command line
  • **stackvis**: generate flame graphs

• Modules:
  • **bunyan** (logging)
  • **restify** (REST/HTTP server, HTTP client)
  • **vasync** (asynchronous control flow)
  • **kang** (expose internal state over HTTP, plus CLI)
  • **dtrace-provider** (application-level probes)
Bonus: native heap analysis

- `pmap -x`: show VA mappings, RSS
- We link with libumem, which has great debugging tools
  - `::findleaks`: finds leaks in native code
  - `::walk umem_alloc_4096`
  - `ptr::whatis`
  - `::walk buftcl | ::bufctl -a PTR -v`
  - `::umastat`
- Example: Wal-Mart memory leak
Native heap tracing

• With DTrace, trace:
  • malloc(3C) / free(3C) / brk(2)
  • operator new / operator delete
• Save a JavaScript stack trace each time
JavaScript heap tracing

- With DTrace, trace `mmap(2)` and `munmap(2)`