Industrial-Grade Node.js

David Pacheco (@dapsays)

NodeSummit 2015

Background

- Joyent runs lots of Node.js
  - Joyent Public Cloud (runs Smart Data Center)
  - Smart Data Center (SDC)
  - Manta

Introduction

- What do we mean by “industrial-grade”?
- How do we build industrial-grade software?

Industrial-grade software

- High quality software (i.e., highly reliable software)

Industrial-grade software

- High quality software (i.e., highly reliable software)
- We don't get there by writing perfect code. We write good code, then we run it, then we find the bugs, and then we fix them.

Industrial-grade software

- High quality software (i.e., highly reliable software)
- We don't get there by writing perfect code. We write good code, then we run it, then we find the bugs, and then we fix them.
- The tools for building industrial-grade software are the tools that help you **find and fix bugs**.

Industrial-grade software

- High quality software (i.e., highly reliable software)
- We don’t get there by writing perfect code. We write good code, then we run it, then we find the bugs, and then we fix them.
- The tools for building industrial-grade software are the tools that help you **find and fix bugs**.
- **Primacy of debuggability**: Debuggability can’t easily be bolted on after-the-fact, but you can do a lot during development to make your program debuggable in production!

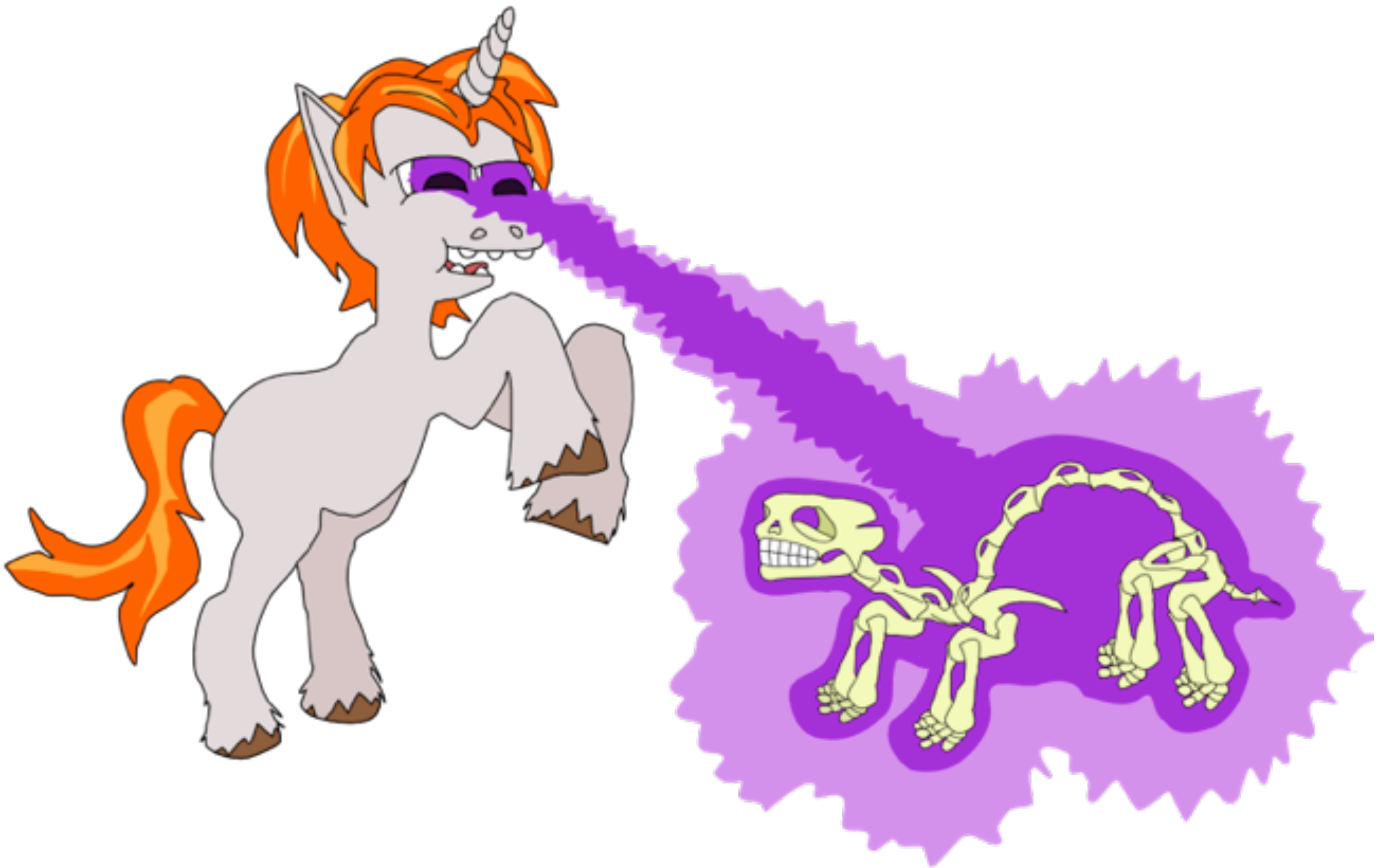
Industrial-grade software

- When we encounter an issue in production, we have two goals:
  - Restore service immediately.
  - Root-cause it *completely* the *first time* it happens.
- These goals can be in tension, but there are techniques to deal with that.

Kinds of software problems

- “What is my program doing?” problems
    - Poor performance (low throughput or high latency)
    - Pathological performance (“what’s it doing?”)
    - Wrong behavior (wrong output)
- Crashes
- Memory problems (leaks, excessive usage)

DTrace



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

DTrace for Node.js

Joyent

- DTrace is a foundation for tons of Node.js observability
  - built-in Node probes: http req/res, GC (see **nhttpsnoop** tool)
  - built-in system probes: memory allocation, syscalls
  - incredibly easy to add your own probes with node-dtrace-provider (e.g., node-restify)
  - **systemic profiling**
- Demos

Example: tracing request latency

```
# /var/tmp/nhttpsnoop -cgsl
TIME                PID PROBE         LATENCY METHOD PATH
[   0.068996]  15832 server ->            - GET     /jobs
[   0.073913]  15832 server <-      4.916ms GET     /jobs
[   0.396989]  16511 client ->            - GET     /configs/65879ef3
[   0.397242]  29441 server ->            - GET     /configs/65879ef3
[   0.409515]  29441 server <-     12.272ms GET     /configs/65879ef3
[   0.409611]  16511 client <-     12.622ms GET     /configs/65879ef3
[   0.411069]  16511 gc     <-      0.863ms -       -
```

Example: garbage collection?

Joyent

```
# ./nhttpsnoop -g -p 7149
...
[189.379996]   7074 gc          1.133ms -       -
[191.113105]   7149 gc        139.936ms -       -
[193.235019]   7149 gc        139.525ms -       -
[194.782425]   7149 gc        142.076ms -       -
[196.355522]   7149 gc        135.985ms -       -
[197.936199]   7149 gc        125.828ms -       -
[197.973465]   7074 gc          1.076ms -       -
[200.649111]   7149 gc        124.679ms -       -
[201.923295]   7149 gc        123.665ms -       -
[203.163419]   7149 gc        124.221ms -       -
[204.634444]   7149 gc        140.286ms -       -
```

Example: restify tracing

```
# ./restify-latency.d -p 25561
^C
ROUTE LATENCY (milliseconds)

            key      min .----------------------------. max        | count
     getconfigs      < 0 :            ▂  ▂▂         ▂: >= 25      | 6
headagentprobes      < 0 : █                          : >= 25      | 5
        listvms      < 0 :           ▁▃ ▁             : >= 25      | 5

HANDLER LATENCY (milliseconds)

                        key      min .----------------------------. max        | count
    ...
    listvms      addProxies      < 0 : █                          : >= 25      | 5
    listvms          bunyan      < 0 : █                          : >= 25      | 5
    listvms      checkMoray      < 0 : █                          : >= 25      | 5
    listvms      checkWfapi      < 0 : █                          : >= 25      | 5
    listvms       handler-0      < 0 : █                          : >= 25      | 5
    listvms         listVms      < 0 :           ▁▃ ▁             : >= 25      | 5
    listvms          loadVm      < 0 : █                          : >= 25      | 5
    listvms     parseAccept      < 0 : █                          : >= 25      | 5
    listvms       parseBody      < 0 : █                          : >= 25      | 5
    listvms       parseDate      < 0 : █                          : >= 25      | 5
    listvms parseQueryString      < 0 : █                          : >= 25      | 5
    listvms        readBody      < 0 : █                          : >= 25      | 5
```

Profiling

- When Node is on-cpu, we use DTrace-based profiling:

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

- We visualize the results with **flame graphs**.
- Demo

Example: on-cpu profiling

Node.js performance

Joyent

- Throughput vs latency
- Useful to divide into **off-cpu** vs. **on-cpu**
- Off-cpu: latency coming from external sources (e.g., database, filesystem, network)
  - to trace: add probes for start/done and trace latency (in a pinch, can also trace libuv)
- On-cpu: latency coming from executing V8 (can be JavaScript or garbage collection)
  - to trace: profile call stacks

Bonus: runtime log snooping

- We use node-bunyan for logging (simple JSON format)
- "trace" and "debug" can be too verbose for production
- But we can get "trace"- and "debug"-level logs of a *running* program (no restart needed) using "bunyan -p", which uses DTrace under the hood.
- Demo

More ways to observe programs

- REPL
  - very useful, but also dangerous
- kang
  - simple library for exposing debug info over HTTP
  - client fetches state from multiple servers

Limitations of runtime observability

- You can only see what's happening right now.
- If you want to debug something that happened before, you have to try to reproduce it. This can lead to expensive try-tracing-this-and-repro-again cycles.
- In production, time spent debugging is downtime!
- You're often at the mercy of the bug reporter (other devs, testers, ops, and other users) for the accuracy and completeness of information

“The postmortem technique”

“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

- Core files: the ultimate REPL.
- Minimally disruptive:
  - Restore service immediately, debug later
  - Can run sophisticated, expensive analysis *offline*
- Get all the facts, not someone's interpretation of their selection of them.
- Solve the problem the first time. Avoid expensive repro cycles. Works in dev, testing, and production!
- Demo

Core files

- Includes **all** of your Node program's state
- To generate a core file on crash, run node with `--abort-on-uncaught-exception`
- To examine a running program, use `gcore(1)`.

Node.js core file debugging

- jsstack: stack trace
- jsprint: print JavaScript objects
- findjsobjects: all objects allocated, by signature
- findjsfunctions: closures
- jsscope: closed-over variables

Memory analysis

- Postmortem approach enables sophisticated memory analysis tools
  - Enumerate and classify all JavaScript objects
  - Enumerate and count all JavaScript closures
- These can be combined with native tools
  - Example: find JS stack that led to a C memory leak

Developing for debugging

Joyent

- Record extra debugging info
  (e.g., timestamps instead of booleans, retry counters)
- Compile everything with **`-fno-omit-frame-pointer`**
- node-vasync: more observable version of “async”
- name prefixes
- javascriptlint

Summary

- Industrial-grade software is highly reliable software. It only gets to that level by finding and fixing the bugs. To do this, we need tools for observing software.
- Runtime Node.js observability tools: dtrace, nhttpsnoop, flame graphs, bunyan, kang, REPL
- Postmortem tools: --abort-on-uncaught-exception, gcore, MDB
- Memory analysis (both JS and native)

Summary: key tools and modules

- Debugging @ Node Dev Center
  https://www.joyent.com/developers/node/debug
- 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)

Industrial-Grade Node.js

David Pacheco (@dapsays)
NodeSummit 2015