Debugging microservices in production

Bryan Cantrill
CTO

bryan@joyent.com
@bcantrill

Debugging in the beginning...

Joyent

Debugging in the beginning...

Joyent

— Sir Maurice Wilkes, 1913 - 2010

Debugging in the beginning...

Joyent

	80	T	46	L
	81	H	45	S
X	82	N	46	~~S~~ L
	83	A	38	L
	84	S	36	L
	85	A	38	L
	86	U	42	L
	87	S	40	L
	88	E	91	S
	89	T	48	L
	90	S	48	L
88 →	91	S	50	L
	92	G	112	S
	93	T		S
	94	A	42	L
	95	T	40	L
	96	E	74	S

— Sir Maurice Wilkes, 1913 - 2010

Debugging in the beginning...

Joyent

	80	T	46	L
	81	H	45	S
X	82	N	46	~~S~~ L
	83	A	38	L
	84	S	36	L
	85	A	38	L
	86	U	42	L
	87	S	40	L
	88	E	91	S
	89	T	48	L
	90	S	48	L
88 →	91	S	50	L
	92	G	112	S
	93	T		S
	94	A	42	L
	95	T	40	L
	96	E	74	S

“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

Debugging

Joyent

- The first non-trivial program for the EDSAC (a program to calculate a table of Airy integrals) had 120 lines and 20 errors — including one not debugged until four decades later!
- This experience remains modern for anyone in software today, and many spend much of their career debugging
- Yet there is little formalized about debugging: few books on it; little research; no conferences — and no university courses!
- Is it any surprise that debugging anti-patterns persist?

Debugging anti-patterns

Joyent

- For too many, debugging is the process of making problems go away rather than understanding the system!
- The view of bugs-as-nuisance has many knock-on effects:
  - Fixes that don’t fix the problem (or introduce new ones!)
  - Bug reports closed out as “will not fix” or “works for me”
  - Users who are told to “restart” or “reboot” or “log out” or anything else that amounts to wishful thinking
- And this is only when the process has obviously failed...

Darker debugging anti-patterns

Joyent

- More insidious effects are felt when the problem *appears* to have been resolved, but hasn't *actually* been fully understood
- These are the fixes that amount to a fresh coat of paint over a crack in the foundation — **and they are worse than nothing**
- Not only do these fixes not actually resolve the problem, they give the engineer a false sense of confidence that spreads virally
- "Debugging" devolves into an oral tradition: folk tales of problems that were made to go away

Thinking methodically

Joyent

- The way we think about debugging is **fundamentally wrong**; we need to think *methodically* about debugging!
- When we think of debugging as the quest for understanding our (misbehaving) systems, it allows us to consider it more abstractly
- Namely, how do we explain the phenomena that affect our world?
- We have found that the most powerful explanations reflect an understanding of underlying structure — beyond *what* to *why*
- This deeper understanding allows us to not only to *explain* but make *predictions*

Predictive power

Joyent

- Valuing predictive power allows us to test our explanations: if our predictions are wrong, our understanding is incomplete
- We can use the understanding from failed predictions to develop *new* explanations and *new* predictions
- We can then test these new predictions to test our understanding
- If all of this is sounding familiar, it's because it's *science* — and the methodical exploration of it is the *scientific method*

The scientific method

Joyent

- The scientific method is to:
  - Make observations
  - Formulate a question
  - Formulate a hypothesis that answers the question
  - Formulate predictions that test the hypothesis
  - Test the predictions by conducting an experiment
  - Refine the hypothesis and repeat as needed

Science, seriously?!

Joyent

Science, seriously.

Joyent

- Software debugging is a *pure distillation* of scientific thinking
- The limitless amount of data from software systems allows experiments in *seconds* instead of weeks/months/years
- The systems we're reasoning about are entirely synthetic, discrete and mutable — we made it, we can understand it
- Software is mathematical machine; the conclusions of software debugging are often mathematical in their unequivocal power!
- Software debugging is *so* pure, it requires us to refine the scientific method slightly to reflect its capabilities...

The software debugging method

Joyent

- Make observations
- Based on observations, formulate a question
- If the question can be answered through subsequent observation, answer the question through observation and refine/iterate
- If the question cannot be answered through observation, make a hypothesis as to the answer and formulate predictions
- If predictions can be tested through subsequent observation, test the predictions through observation and refine/iterate
- Otherwise, test predictions through experiment and refine/iterate

Observation is the heart of debugging!

Joyent

- The essence — and art! — of debugging software is making observations and asking questions, **not** formulating hypotheses!
- Observations are *facts* — they constrain hypotheses in that any hypothesis contradicted by facts can be summarily rejected
- As facts beget questions which beget observations and more facts, hypotheses become more tightly constrained — like a cordon being cinched around the truth
- Or, in the words of Sir Arthur Conan Doyle's Sherlock Holmes, "when you have eliminated all which is impossible, then whatever remains, however improbable, must be the truth"

Making the hypothetical leap

Joyent

- Once observation has sufficiently narrowed the gap between what is known and what is wrong, a *hypothetical leap* should be made
- Debugging is inefficient when this leap is made too early — like making a specific guess too early in Twenty Questions
- A hypothesis is only as good as its ability to form a *prediction*
- A prediction should be tested with either subsequent observation or by conducting an experiment
- If the prediction proves to be incorrect, understanding is incomplete; the hypothesis must be rejected — or refined

Experiments in software

Joyent

- A beauty of software is that it is highly amenable to experiment
- Many experiments are programs — and the most satisfying experiments test predictions about how failure can be induced
- Many “non-reproducible” problems are merely unusual!
- Debugging a putatively non-reproducible problem to the point of a reproducible test case is a joy unique in software engineering

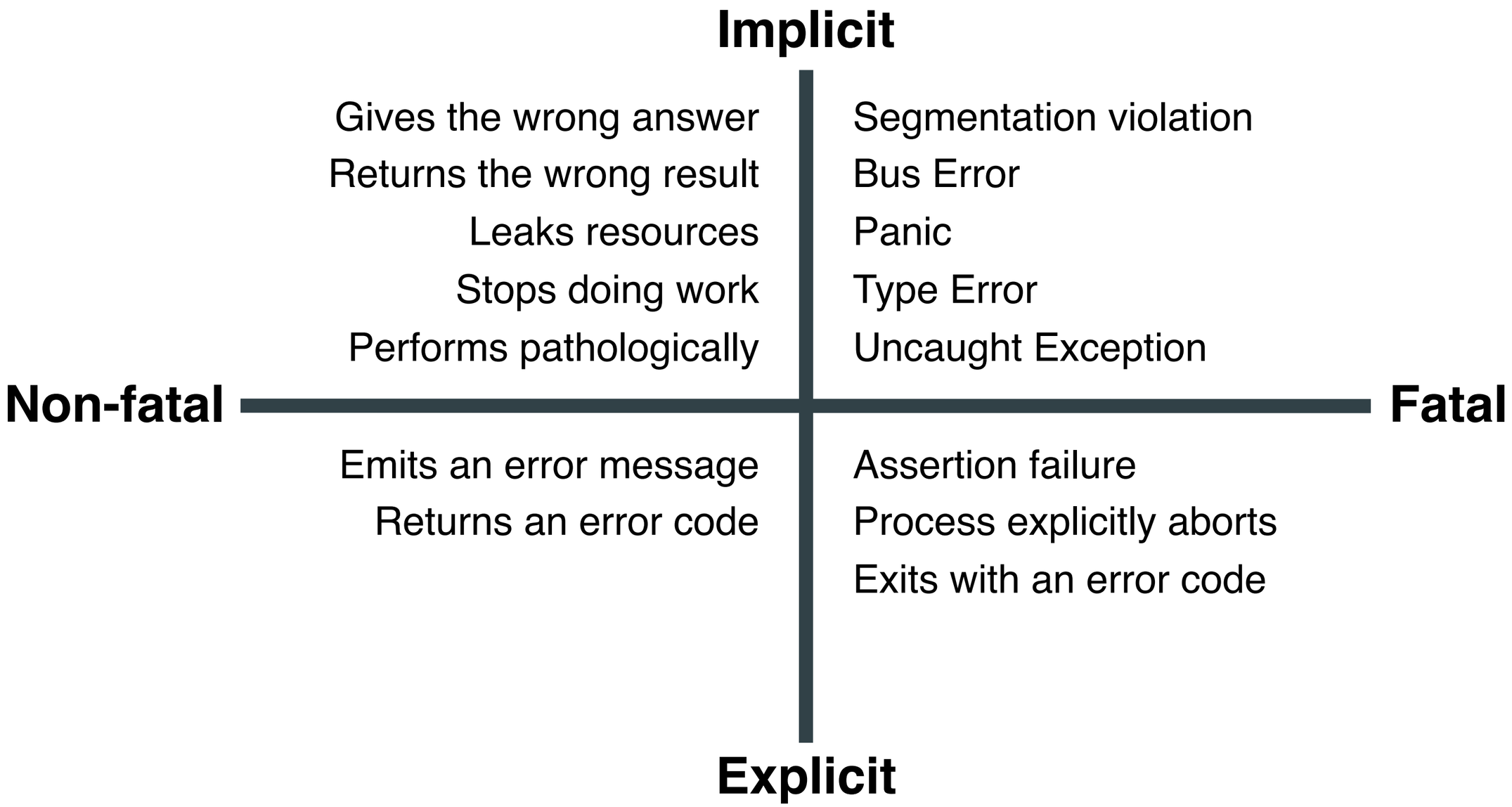
Software debugging in practice

Joyent

- The specifics of observation depends on the nature of the failure
- Software has different kinds of *failure modes*:
  - Fatal failure (segmentation violation, uncaught exception)
  - Non-fatal failure (gives the wrong answer, performs terribly)
  - Explicit failure (assertion failure, error message)
  - Implicit failure (cheerfully does the wrong thing)

Taxonomizing software failure

Joyent



Microservices prehistory

Joyent

- The late 1990s saw the rise of *three-tier architectures* consisting of presentation, application logic and data tiers
- Many names for roughly the same notion: “Service-oriented architecture”, “Model/View/Controller”, etc.
- The AJAX+REST revolution of the mid-2000s gave rise to true *web applications* in which application logic could live on the edge
- Led to some broader architectural questioning...

Post-AJAX questions

Joyent

- Why should HTTP be restricted to the web?
- Why should REST be restricted to web apps?
- Instead of having one monolithic architecture, why not have a series of (smaller) services that merely did one thing well?
- In case this sounds vaguely familiar...

The Unix Philosophy

Joyent

- The Unix philosophy, as articulated by Doug McIlroy:
  - Write programs that do one thing and do it well
  - Write programs to work together
  - Write programs that handle text streams, because that is a universal interface
- The single most important revolution in software systems thinking!
- Applying it to HTTP-based services...

Microservices

Joyent

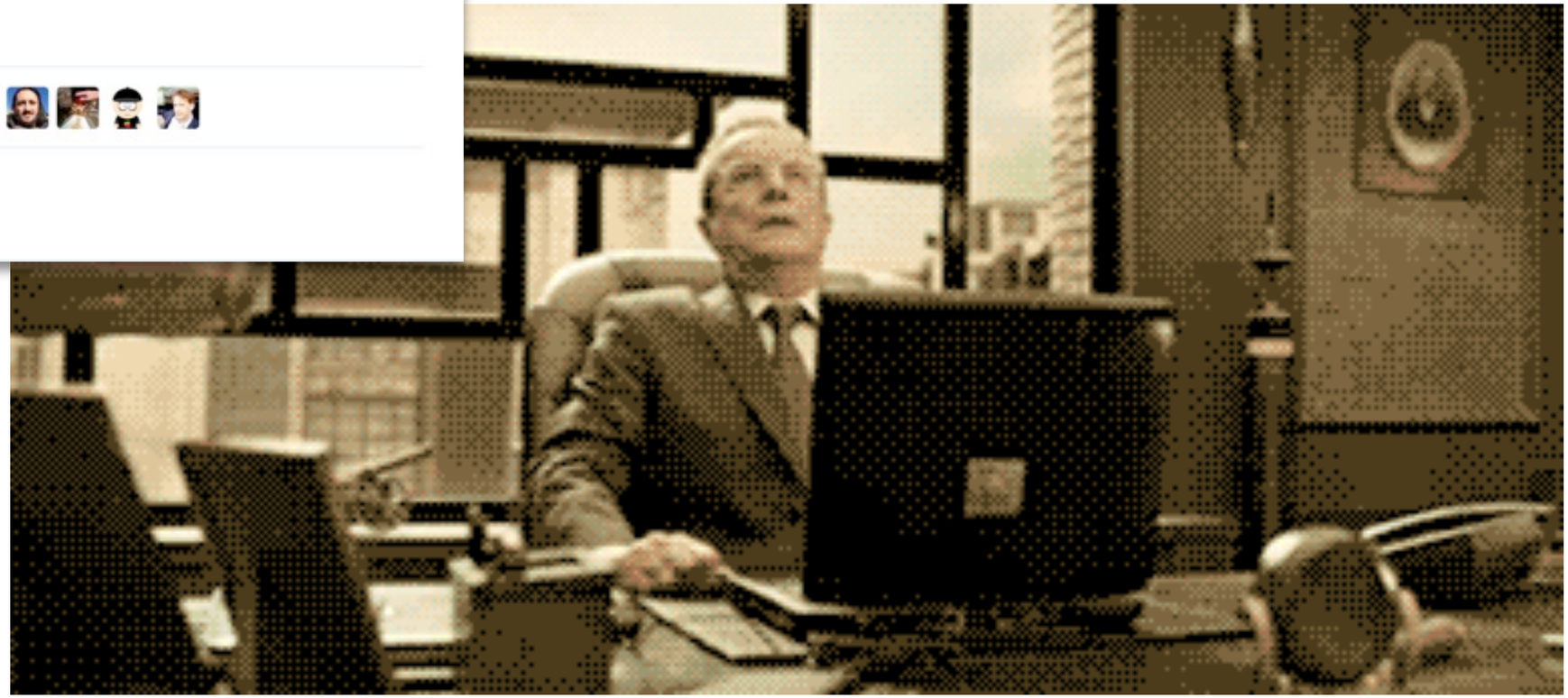
- *Microservices* do one thing, and strive to do it well
- Replace a small number of monoliths with many services that have well-documented, small HTTP-based APIs
- Larger systems can be composed of these smaller services
- While the trend it describes is real, the term “microservices” isn’t without its controversy...

Microservices

Joyent

Microservices

Joyent



Debugging microservices

Joyent

- Veteran nerd rage may be being provoked by proponents of microservices not fully appreciating the risks…
- Microservices turn a *monolithic system* into a *distributed* one
- While resilient to certain classes of *force majeure* failures, distributed systems remain vulnerable to software defects
- Distributed systems are infamously nasty to debug — not least because **they often must be debugged in production**

Microservices in production

Joyent

- Microservices are tautologically small — they don't need their own dedicated physical hardware, or even dedicated *virtual* hardware!
- Microservices are a particularly good fit for *containers*, virtual OS instances pioneered by FreeBSD jails and Solaris zones

Virtualization and Namespace Isolation in Solaris (PSARC/2002/174)

This project introduces Solaris *zones*. Zones provide a means of virtualizing operating system services, allowing one or more processes to run in isolation from other activity on the system. This isolation prevents processes running within a given zone from monitoring or affecting processes running in other zones. A zone is a "sandbox" within which one or more applications can run without affecting or interacting with the rest of the system. It also provides an abstraction layer that separates applications from physical attributes of the machine on which they are deployed, such as physical device paths and network interface names.

Containers at Joyent

- Joyent runs OS containers in the cloud via SmartOS — and we have run containers in multi-tenant production since ~2006
- Adding support for hardware-based virtualization circa 2011 strengthened our resolve with respect to OS-based virtualization
- OS containers are lightweight and efficient — which is especially important as services become smaller and more numerous: overhead and latency become increasingly important!
- We emphasized their operational characteristics — performance, elasticity, tenancy — and for many years, we were a lone voice...

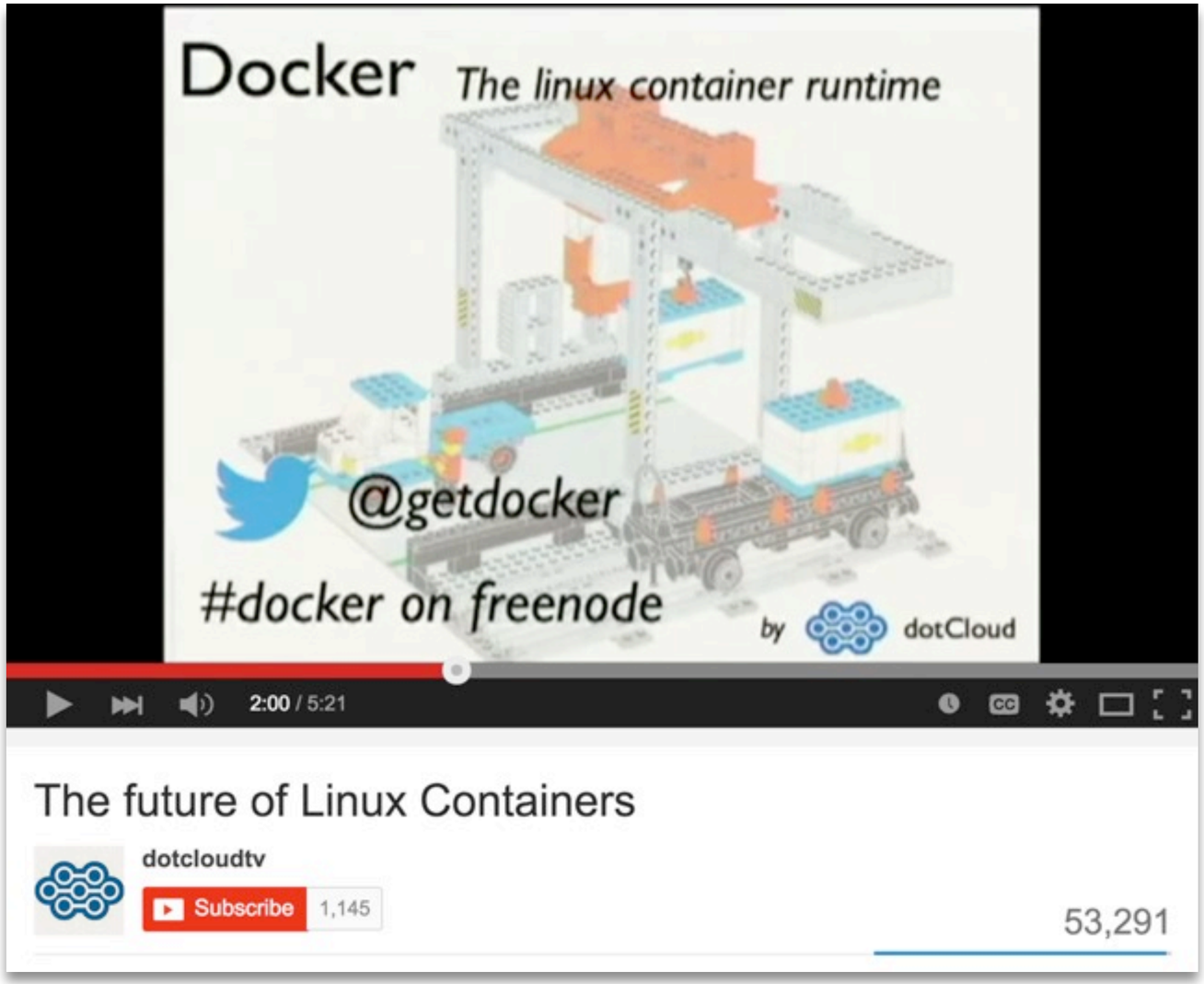
Containers as PaaS foundation?

Joyent

- Some saw the power of OS containers to facilitate up-stack platform-as-a-service abstractions
- For example, dotCloud — a platform-as-a-service provider — built their PaaS on OS containers
- Struggling as a PaaS, dotCloud pivoted — and open sourced their container-based orchestration layer...

...and Docker was born

Joyent



Docker revolution

Joyent

- Docker has used the rapid provisioning + shared underlying filesystem of containers to allow developers to think *operationally*
- Developers can encode deployment procedures via an *image*
- Images can be reliably and reproducibly deployed as a container
- Images can be quickly deployed — and re-deployed
- Docker complements the small-system ethos of microservices!

Docker at Joyent

Joyent

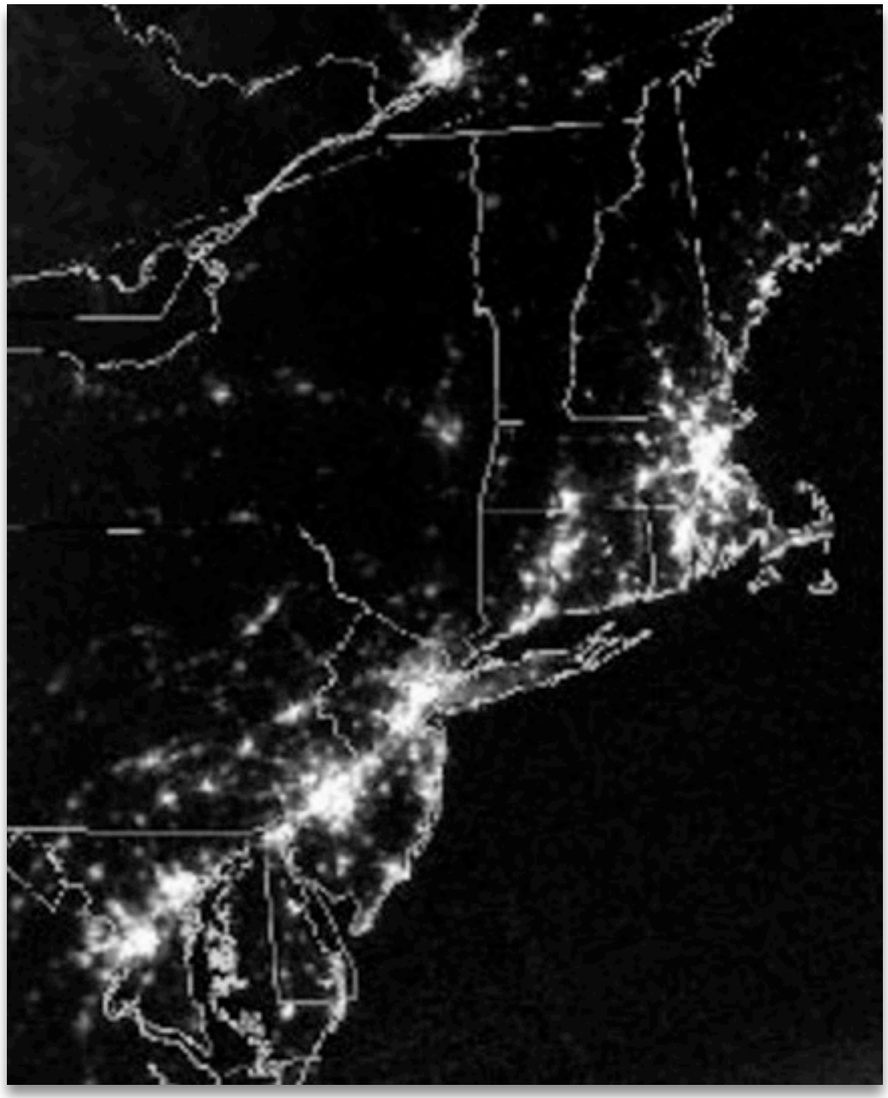
- We wanted to create a best-of-all-worlds platform: the developer ease of Docker on the production-grade substrate of SmartOS
- We developed a Linux system call interface for SmartOS, allowing SmartOS to run Linux binaries at bare-metal speed
- In March 2015, we introduced **Triton**, our (open source!) stack that deploys Docker containers directly on the metal
- Triton virtualizes the notion of a Docker host (i.e., "`docker ps`" shows all of one's containers datacenter-wide)
- Brings full debugging (DTrace, MDB) to Docker containers

When microservices fail?

Joyent

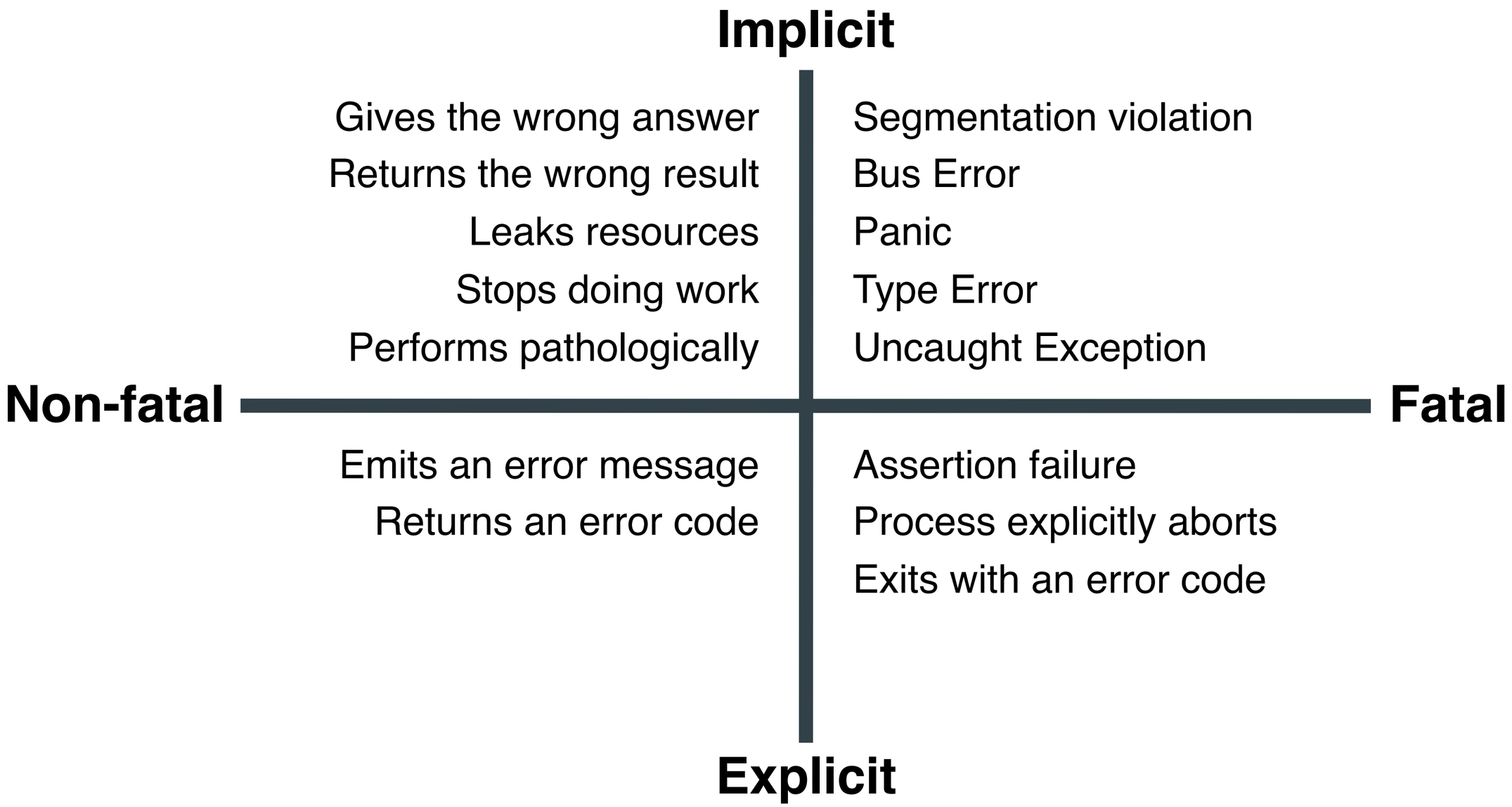
A more apt metaphor...

Joyent



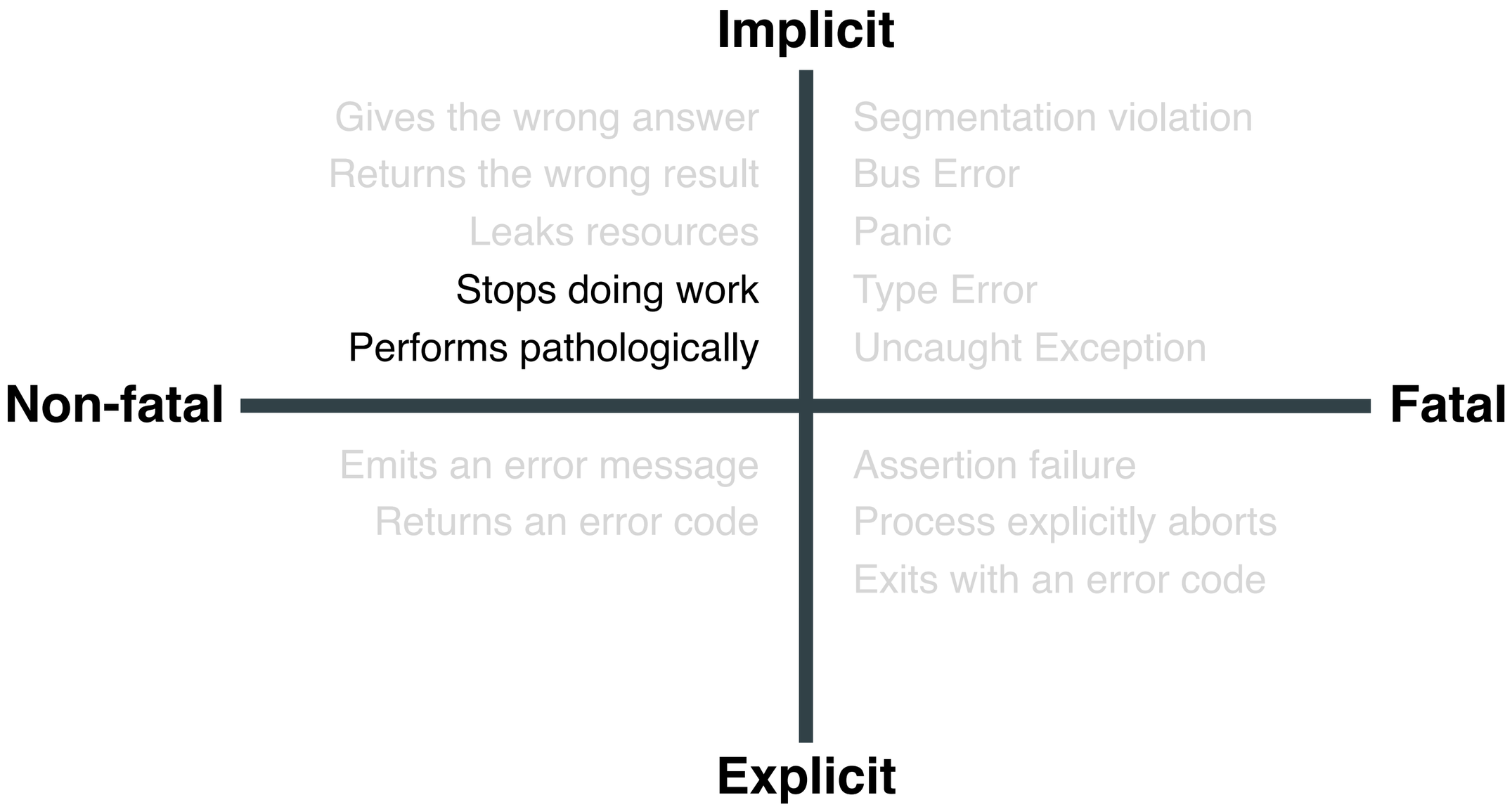
Microservice failure modes

Joyent



Cascading microservice failure modes

Joyent



Debugging fatal failure

Joyent

- When software fails fatally, we know that the software itself is broken — its state has become inconsistent
- By saving in-memory state to stable storage, the software can be debugged *postmortem*
- To debug, one starts with the invalid state and reasons backwards to discover a transition from a valid state to an invalid one
- This technique is so old, that the terms for this saved state dates back to the dawn of the computing age: a *core dump*
- Not as low-level as the name implies! Modern high-level languages (e.g., node.js and Go) allow postmortem debugging!

Debugging fatal failure: microservices

Joyent

- Postmortem analysis lends itself very well to microservices:
  - There is no run-time overhead; overhead (such as it is) is only at the time of death
  - The microservice/container can be safely (automatically!) restarted; the core dump can be analyzed asynchronously
  - Tooling need not be in container, can be made arbitrarily rich
- In Triton, all core dumps are automatically stored and then uploaded into a system that allows for analysis, tagging, etc.
- This has been invaluable for debugging our own services!

Debugging non-fatal failure

Joyent

- There is a solace in fatal failure: it always represents a software defect at some level — and the inconsistent state is *static*
- Non-fatal failure can be more challenging: the state is *valid* and *dynamic* — it's difficult to separate symptom from cause
- Non-fatal failure must still be understood *empirically*!
- Debugging *in vivo* requires that data be extracted from the system — either of its own volition (e.g., via logs) or by coercion (e.g., via instrumentation)

Debugging explicit, non-fatal failure

Joyent

- When failure is explicit (e.g., an error or warning message), it provides a very important data point
- If failure is non-reproducible or otherwise transient, analysis of explicit software activity becomes essential
- Action in one container will often need to be associated with failures in another
- Especially for distributed systems, this becomes *log analysis*, and is an essential forensic tool for understanding explicit failure
- Essential observation: a time line of events!

Debugging implicit, non-fatal failure

Joyent

- Problems that are both implicit *and* non-fatal represent the most time-consuming, most difficult problems to debug because the system must be understood against its will
  - Wherever possible make software explicit about failure!
  - Where errors are *programmatic* (and not *operational*), they should always induce fatal failure!
- Microservices break at the boundaries: two services each think that they are operating correctly, but together they're broken
- Data must be coerced from the system via *instrumentation*

Instrumenting production systems

Joyent

- Traditionally, software instrumentation was hard-coded and static (necessitating software restart or — worse — recompile)
- Dynamic system instrumentation was historically limited to system call table (strace/truss) or packet capture (tcpdump/snoop)
- Effective for some problems, but a poor fit for *ad hoc* analysis
- In 2003, Sun developed *DTrace*, a facility for arbitrary, dynamic instrumentation of production systems that has since been ported to Mac OS X, FreeBSD, NetBSD and (to a degree) Linux
- DTrace has inspired dynamic instrumentation in other systems (see @brendangregg's talk!)

Instrumenting Docker containers

Joyent

- In Docker, instrumentation is a challenge as containers may not include the tooling necessary to understand the system
- Docker host-based techniques for instrumentation may be tempting, but they should be considered an anti-pattern!
- DTrace has a privilege model that allows it to be safely (and usefully) used from within a container
- In Triton, DTrace is available from within every container — one can "`docker exec -it bash`" and then debug interactively

Instrumenting node.js-based microservices

Joyent

- We have invested heavily in node.js-based infrastructure to allow us to meaningfully instrument microservices in production:
  - We developed *Bunyan*, a logging facility for node.js that includes DTrace support
  - Added DTrace support for node.js profiling
- An essential vector for iterative observation: turning up the logging level on a running microservice!

Debugging microservices in production

Joyent

- Debugging methodically requires us to shift our thinking — and learn how to carefully *observe* the systems we build
- Different types of failures necessitate different techniques:
  - Fatal failure is best debugged via postmortem analysis — which is particular appropriate in an all-container world
  - Non-fatal failure necessitates log analysis and dynamic instrumentation
- The ability to debug problems in production is essential to successfully deploy and scale microservices!