Running Aground: Debugging Docker in production

Bryan Cantrill (@bcantrill), CTO, Joyent

dockercon 15

SF JUNE 22-23

The Docker revolution

- While OS containers have been around for over a decade, Docker has brought the concept to a much broader audience
- Docker has used the rapid provisioning and shared filesystem of containers to allow developers to think *operationally*
  - Deployment procedures can be encoded via an image
  - Images can be reliably and reproducibly deployed as containers
- **Docker is doing to apt what apt did to tar**

Docker at Joyent

- At Joyent, we have run SmartOS-based containers on the metal and in multi-tenant production since ~2006
- 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)

Docker + microservices

- Docker is particular apt at deploying *microservices*: small, well-defined services that do one thing and do it well
- While the term provokes some degree of nerd rage, it is merely a new embodiment of an *old idea*: the Unix Philosophy
- What does the container + microservices revolution mean for how we *debug* programs and systems?

Debugging: An even older idea

Debugging: An even older idea

Sir Maurice Wilkes, 1913 - 2010

Debugging: An even older idea

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

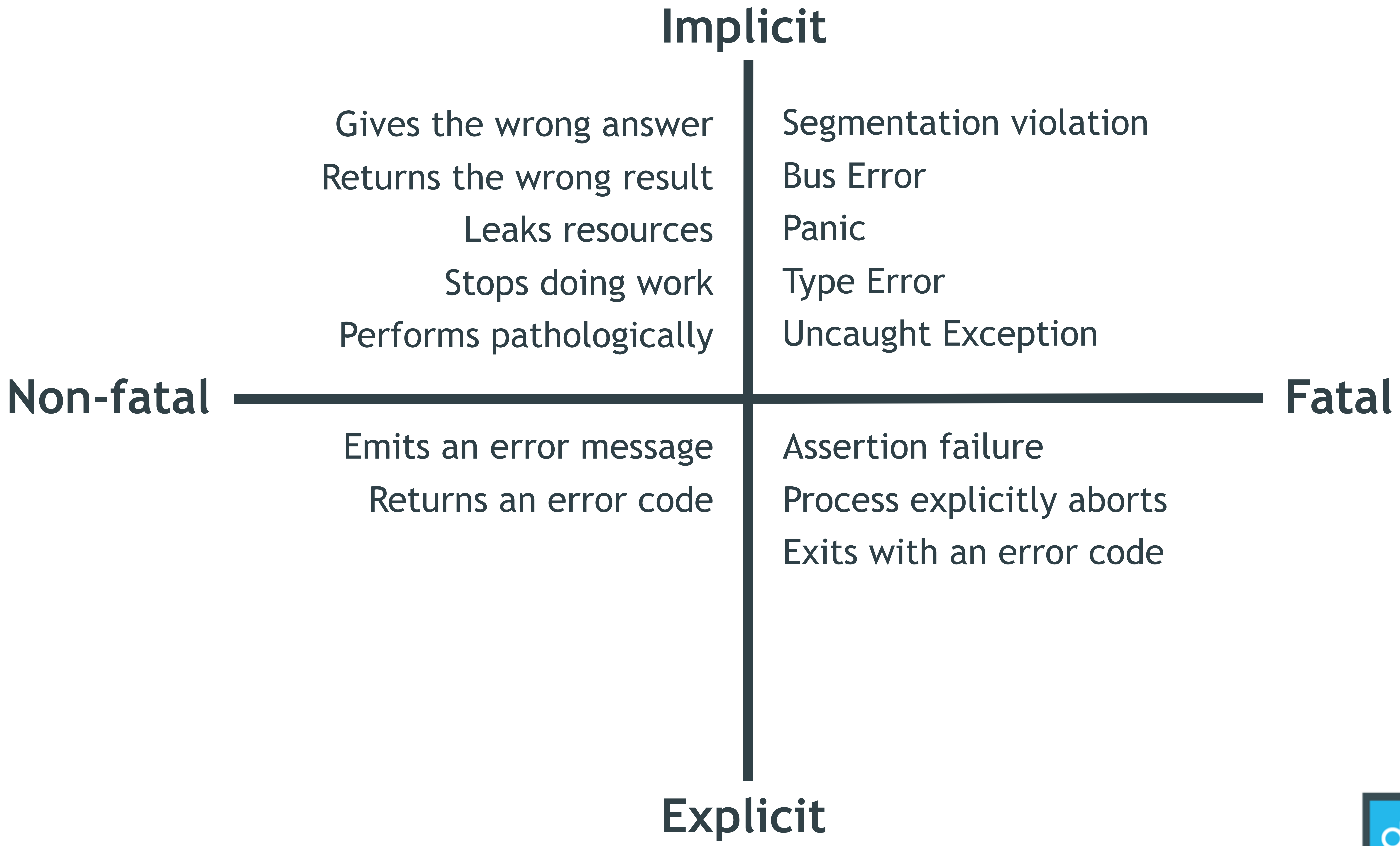
Debugging Docker

- When deploying Docker + microservices, there is an unstated truth: **you are developing a distributed system**
- While more resilient to certain classes of *force majeure* failure, distributed systems remain vulnerable to software defects
- Worse, distributed systems are harder to debug — and are more likely to exhibit behavior non-reproducible in development
- Docker forces us to change the way we debug systems: **we must debug not in terms of *sick pets* but rather *sick cattle***

Software failure

- Different failure modes have different implications for debugging!
- And software has many different 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



Debugging fatal failure

- 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) have capabilities for postmortem debugging

Debugging fatal failure: Containers

- Postmortem analysis lends itself very well to the container model:
  - There is no run-time overhead; overhead (such as it is) is only at the time of death
  - The container can be safely (automatically!) restarted; the core dump can be analyzed asynchronously
  - Debugging tooling can be made arbitrarily rich, as it need *not* exist within the failing container

Core dump management in Docker

- 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!
- Outside of Triton, the lack of container awareness around core_pattern in the Linux kernel is problematic for Docker: core dumps from Docker are still a bit rocky (viz. docker#11740)
- Docker-based core dump management (e.g., "`docker dumps`"?) would be a welcome addition!

Debugging non-fatal failure

- 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* — and it can be 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

- 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
- For modern software, this becomes *log analysis*, and is an essential forensic tool for understanding explicit failure

Log management in Docker

- "`docker logs`" is fine when the problem is simple — but more complicated issues will require more sophisticated analysis
- Deeper analysis requires logs be moved out of a container
- Docker is not prescriptive about how this is done, and there are many ways to do it:
  - Logs can be shipped from a process within the container
  - Logs can be pulled from a container that is sharing a volume
- Log management techniques that rely on Docker host manipulation should be considered an anti-pattern!

Aside: Docker host anti-patterns

- In the traditional Docker model, Docker hosts are virtual machines to which containers are directly provisioned
- It may become tempting to manipulate Docker hosts directly, but doing this **entirely compromises the Docker security model**
- Worse, compromising the security model creates a **VM dependency** that makes bare-metal containers impossible
- And ironically, **Docker hosts become pets**: the reasons for backdooring through the Docker host come to resemble the arguments made by those who resist containerization entirely!

Debugging implicit, non-fatal failure

- 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!
- Data must be coerced from the system via *instrumentation*

Instrumenting production systems

- 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 software in other systems (see Brendan Gregg’s talks for details)

Instrumenting Docker containers

- In Docker, instrumentation is a challenge as containers may not include the tooling necessary to understand the system
- Host-based techniques for instrumentation may be tempting, but (again!) 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

Debugging Docker in production

- Debugging Docker in production requires us to shift our thinking from sick pets to sick cattle
- 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 production problems is essential for Docker to leap the chasm into broad production deployment!

Thank you

Bryan Cantrill

@bcantrill, bryan@joyent.com