And It All Went Horribly Wrong: Debugging Production Systems

Bryan Cantrill
VP, Engineering

bryan@joyent.com
@bcantrill

In the beginning...

In the beginning...

Joyent

Sir Maurice Wilkes, 1913 - 2010

In 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

Debugging through the ages

Joyent

- As systems had more and more demands placed upon them, we became better at debugging their failures...
- ...but as these systems were replaced (disrupted) by faster (cheaper) ones, debuggability often regressed
- At the same time, software has been developed at a higher and higher layer of abstraction — and accelerated by extensive use of componentization
- The high layers of abstraction have made it easer to get the system initially working (develop) — but often harder to understand it when it fails (deploy + operate)
- Production systems are *more* complicated and *less* debuggable!

So how have we made it this far?

- We have architected to survive component failure
- We have carefully considered *state* — leaving tiers of the architecture stateless wherever possible
- Where we have state, we have carefully considered *semantics*, moving from ACID to BASE semantics (i.e., different CAP trade-offs) to increase availability
- ...and even ACID systems have been made more reliable by using redundant components
- Clouds (especially unreliable ones) have expanded the architectural imperative to survive *datacenter* failure

Do we still need to care about failure?

- Software engineers should not be fooled by the rise of the putatively reliable distributed system; single component failure still has significant cost:
  - **Economic cost**: the system has fewer available resources with the component in a failed state
  - **Run-time cost**: system reconstruction or recovery often induces additional work that can degrade performance
- Most dangerously, single component failure puts the system in a *more vulnerable mode* whereby further failure becomes more likely
- This is *cascading failure* — and it is what induces failure in mature, reliable systems

Disaster Porn I

Joyent

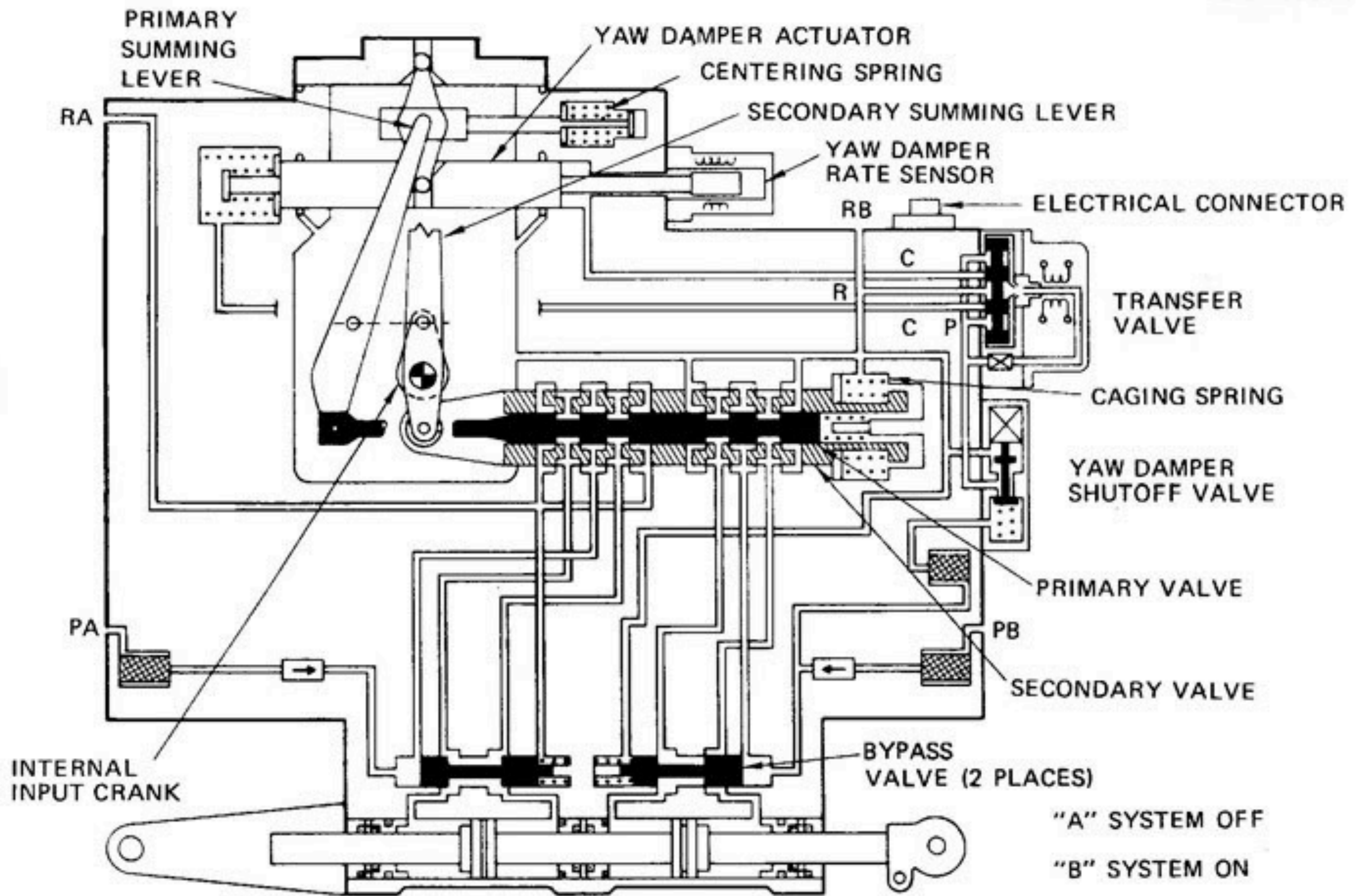
Wait, it gets worse

Joyent

- This assumes that the failure is fail-stop — a failed drive, a panicked kernel, a seg fault, an uncaught exception
- If the failure is transient or byzantine, single component failure can *alone* induce system failure
- Monitoring attempts to get at this by establishing rich liveness criteria for the system — and allowing the operator to turn transient failure into fatal failure...
- ...but if monitoring becomes too sophisticated or invasive, it risks becoming so complicated as to compound failure

Disaster Porn II

Joyent



Debugging in the modern era

Joyent

- Failure — even of a single component — *erodes the overall reliability* of the system
- When single components fail, *we must understand why* (that is, we must debug them), and we must fix them
- We must be able to understand both fatal (fail-stop) failures *and* (especially) transient failures
- We must be able to diagnose these *in production*

Debugging fatal component failure

Joyent

- When a software component fails fatally (e.g., due to dereferencing invalid memory or a program-induced abort) its state is *static* and *invalid*
- By saving this state (e.g., DRAM) to stable storage, the component can be debugged *postmortem*
- One starts with the invalid state and proceeds backwards to find the transition from a valid state to an invalid one
- This technique is so old, that the term for this state dates from the dawn of the computing age: a *core dump*

Postmortem advantages

Joyent

- There is no run-time system overhead — cost is only induced when the software has fatally failed, and even then it is only the cost of writing state to stable storage
- Once its state is saved for debugging, there is nothing else to learn from the component's failure *in situ*; it can be safely restarted, minimizing downtime without sacrificing debuggability
- Debugging of the code dump can occur asynchronously, in parallel, and arbitrarily distant in the future
- Tooling can be made extraordinarily rich, as it need not exist on the system of failure

Disaster Porn III

Joyent

Postmortem challenges

Joyent

- Must have the mechanism for saving state on failure
- Must record sufficient state — which must include program text as well as program data
- Must have sufficient state present in DRAM to allow for debugging (correctly formed stacks are a must, as is the symbol table; type information is invaluable)
- Must manage state such that storage is not overrun by a repeatedly pathological system
- These challenges are real but surmountable — and several open source systems have met them...

Postmortem debugging: MDB

Joyent

- For example, MDB is the debugger built into the open source illumos operating system (a Solaris derivative)
- MDB is modular, with a plug-in architecture that allows for components to deliver custom debugger support
- 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

Postmortem debugging

Joyent

- Postmortem debugging is well advanced for native code — but much less developed for dynamic environments like Java, Python, Ruby, JavaScript, Erlang, etc.
- Of these, only Java has made a serious attempt at postmortem debugging via the jdb(1) tool found in HotSpot VM — but it remains VM specific
- If/as dynamic environments are used for infrastructural software components, it is critical that they support postmortem debugging as a first-class operation!
- In particular, at Joyent, we're building many such components in node.js...

Aside: node.js

Joyent

- node.js is a JavaScript-based framework (based on Google's V8) for building event-oriented servers:

```
var http = require('http');

http.createServer(function (req, res) {
        res.writeHead(200, {'Content-Type': 'text/plain'});
        res.end('Hello World\n');
}).listen(8124, "127.0.0.1");

console.log('Server running at http://127.0.0.1:8124!');
```

- node.js makes it very easy to build a reliable, event-oriented networking services

Postmortem debugging: node.js

Joyent

- Debugging a dynamic environment requires a high degree of VM specificity in the debugger…
- ...but we can leverage MDB's module-oriented nature to do this somewhat cleanly with a disjoint V8 module
- Joyent's Dave Pacheco has built MDB dmods to be able to symbolically dump JavaScript stacks and arguments from an OS core dump:
  - `::jsstack` prints out a JavaScript stack trace
  - `::jsprint` prints out a JavaScript heap object from its C++ (V8) handle
- Details:

  http://dtrace.org/blogs/dap/2011/10/31/nodejs-v8-postmortem-debugging/

Postmortem debugging: node.js

Joyent

- node.js postmortem debugging is still nascent; there's much more to do here
- For example, need a way to induce an abort(3C) from JavaScript to allow program-induced core dumps…
- ...but it's still incredibly useful on gcore(1)-generated core dumps
- We've already used it to nail a bug that was seen exactly twice over the course of the past year — and only in production!

Debugging transient component failure

Joyent

- Despite its violence, fatal component failure can be dealt with architecturally and (given proper postmortem debugging support) be root-caused from a single failure
- Non-fatal component failure is much more difficult to compensate for — and much more difficult to debug!
- State is *dynamic* and *valid* — it's hard to know where to start, and the system is still moving!
- When non-fatal pathologies cascade, it is difficult to sort symptom from cause — you are physician, not scientist
- This is *Leventhal's Conundrum*: given the hurricane, where is the butterfly?

Disaster Porn IV

Joyent

DTrace

Joyent

- Facility for dynamic instrumentation of production systems originally developed circa 2003 for Solaris 10
- Open sourced (along with the rest of Solaris) in 2005; subsequently ported to many other systems (MacOS X, FreeBSD, NetBSD, QNX, nascent Linux port)
- Support for arbitrary actions, arbitrary predicates, *in situ* data aggregation, statically-defined instrumentation
- Designed for safe, *ad hoc* use in production: concise answers to arbitrary questions
- Early on in DTrace development, it became clear that the most significant non-fatal pathologies were high in the stack of abstraction...

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

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 in other environments

Joyent

- This technique has been generalized by Chris Andrews in his node-dtrace-provider npm module:

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

- Chris has also done this for Ruby (ruby-dtrace) and Perl (Devel::DTrace::Provider)
- Neither technique addresses the problem of associating in-kernel events with their user-level (dynamic) context

Extending DTrace into the VMs

Joyent

- To allow DTrace to meaningfully understand VM state from probe context, we introduced the notion of a *helper* — programmatic logic that is attached to the VM itself
- For the stack helper, a VM defines — in D — the logic to get from frame pointers to a string that names the frame
- Must run in the kernel, in probe context — brutally hard to program
- This was done for Java initially, but has also been done for Python by John Levon and node.js by Dave Pacheco

Your DTrace fell into my MDB!

- DTrace data can be recorded to a ring buffer and recovered postmortem after system failure via MDB
- Conversely, DTrace can be used to turn transient failure into fatal failure via its raise() and panic() actions
- DTrace can also be used to stop() a process, which can then be gcore(1)'d and prun(1)'d
- Allows one to get a precisely defined static snapshot of an otherwise dynamic problem
- More generally, using postmortem techniques together with dynamic instrumentation gives one much more latitude in attacking either variant of system pathology!

Thank you!

- Dave Pacheco (@dapsays) for node.js/V8 postmortem debugging work, the V8 ustack helper and his excellent ACM Queue article on postmortem debugging:
  http://queue.acm.org/detail.cfm?id=2039361
- Chris Andrews (@chrisandrews) for libusdt and its offspring: node-dtrace-provider, ruby-dtrace and perl-dtrace
- John Levon (@johnlevon) for the Python ustack helper
- Scott Fritchie (@slfritchie) for his recent work on Erlang support for DTrace
- Adam Leventhal (@ahl) for his conundrum