



# Real-time node.js: Instrumentation, Visualization & Debugging

**Bryan Cantrill**

*SVP, Engineering*

**bryan@joyent.com**

**@bcantrill**

- The term has enjoyed some popularity, but there is clearly confusion about the definition of “real-time”
- A *real-time system* is one in which the correctness of the system is relative to its timeliness
- A *hard* real-time system is one which the latency constraints are rigid: violation constitutes total system failure (e.g., an actuator on a physical device)
- A *soft* real-time system is one in which latency constraints are more flexible: violation is undesirable but non-fatal (e.g., a video game or MP3 player)
- Historically, the only real-time aspect of the web has been in some of its *static* content (e.g. video, audio)

- The rise of mobile + HTML5 has given rise to a new breed of web application: ones in which *dynamic* data has real-time semantics
- These data-intensive real-time applications present new semantics for web-facing applications
- **Data-intensive real-time** gives rise to a new paradigm: CRUD, ACID, BASE, CAP — meet DIRT!

- DIRTy applications tend to have the *human in the loop*
  - Good news: deadlines are *soft* — microseconds only matter when they add up to tens of milliseconds
  - Bad news: because humans are in the loop, demand for the system can be *non-linear*
- One must deal not only with the traditional challenge of scalability, but also the challenge of a real-time system!

- Embedded real-time systems are sufficiently controlled that latency bubbles can be architected away
- Web-facing systems are far too sloppy to expect this!
- Focus must shift from preventing latency bubbles to preventing latency bubbles from *cascading*
- Operations that can induce latency (network, I/O, etc.) must not be able to take the system out with them!
- Implies purely asynchronous and evented architectures, which are notoriously difficult to implement...

- node.js is a JavaScript-based framework 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 is a confluence of three ideas:
  - JavaScript's rich support for asynchrony (i.e. closures)
  - High-performance JavaScript VMs (e.g. V8)
  - The system abstractions that God intended (i.e. UNIX)
- Because everything is asynchronous, node.js is ideal for delivering scale in the presence of long-latency events!

- As the correctness of the system is its timeliness, we must be able to *measure* the system to *verify* it
- In a real-time system, *it does not make sense to measure operations per second!*
- The *only* metric that matters is latency
- This is dangerous to distill to a single number; the *distribution* of latency over time is essential
- This poses both instrumentation and visualization challenges!



- Instrumenting for latency requires modifying the system *twice*: as an operation starts and as it finishes
- During an operation, the system must track — on a *per-operation* basis — the start time of the operation
- Upon operation completion, the resulting stored data cannot be a scalar — the *distribution* is essential when understanding latency
- Instrumentation must be systemic; must be able to reach to the *sources* of latency deep within the system
- These constraints eliminate static instrumentation; we need a better way to instrument the system

- 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, Linux port in development)
- 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
- Particularly well suited to real-time: the original design center was the understanding of latency bubbles

- 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

- 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
- We use USDT is-enabled probes to minimize disabled probe effect once in C++
- If (and only if) the probe is enabled, we prepare a structure for the kernel that allows for translation into a structure that is familiar to node programmers

- Example one-liners:

```
dtrace -n 'node*:::http-server-request{
    printf("%s of %s from %s\n", args[0]->method,
        args[0]->url, args[1]->remoteAddress)}'
```

```
dtrace -n http-server-request'{@[args[1]->remoteAddress] = count()}'
```

```
dtrace -n gc-start'{self->ts = timestamp}' \
    -n gc-done'/self->ts/{@ = quantize(timestamp - self->ts)}'
```

- A script to measure HTTP latency:

```
http-server-request
{
    self->ts[args[1]->fd] = timestamp;
}
```

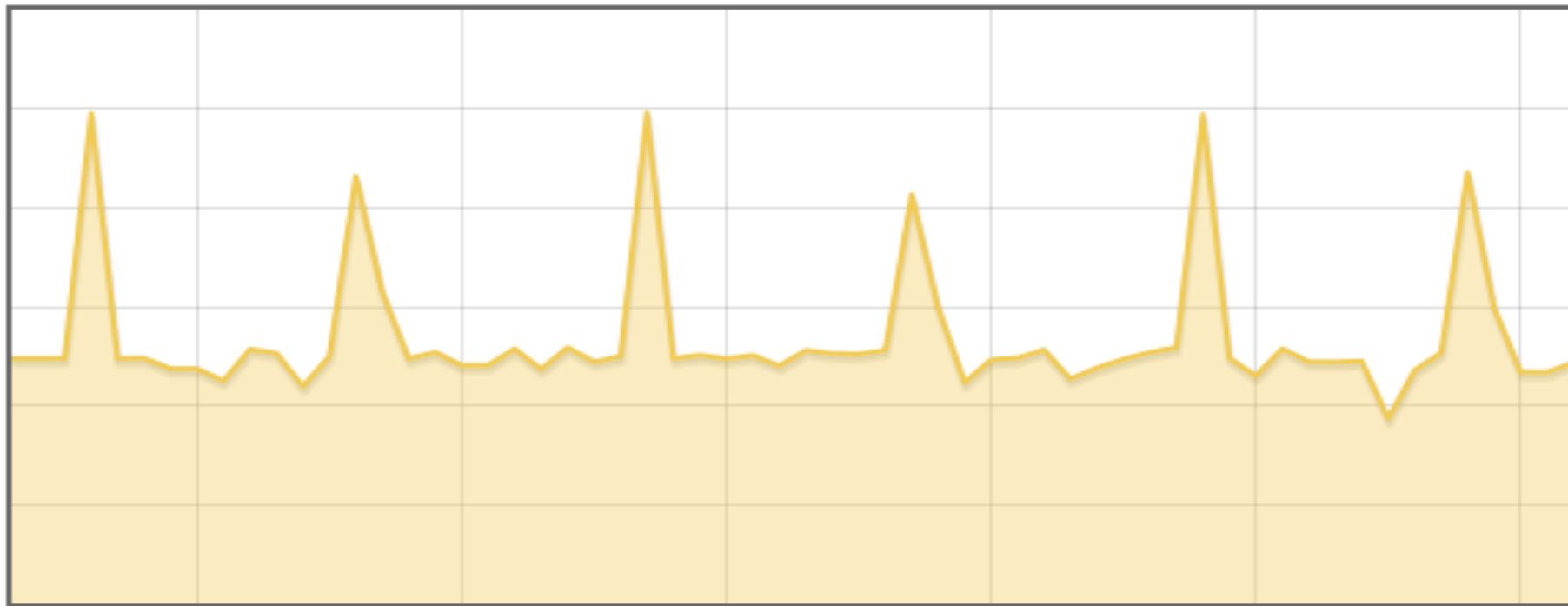
```
http-server-response
/self->ts[args[0]->fd]/
{
    @[zonename] = quantize(timestamp - self->ts[args[0]->fd]);
}
```

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

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

- Used by Joyent's Mark Cavage in his Idap.js to measure and validate operation latency
- But how to visualize operation latency?

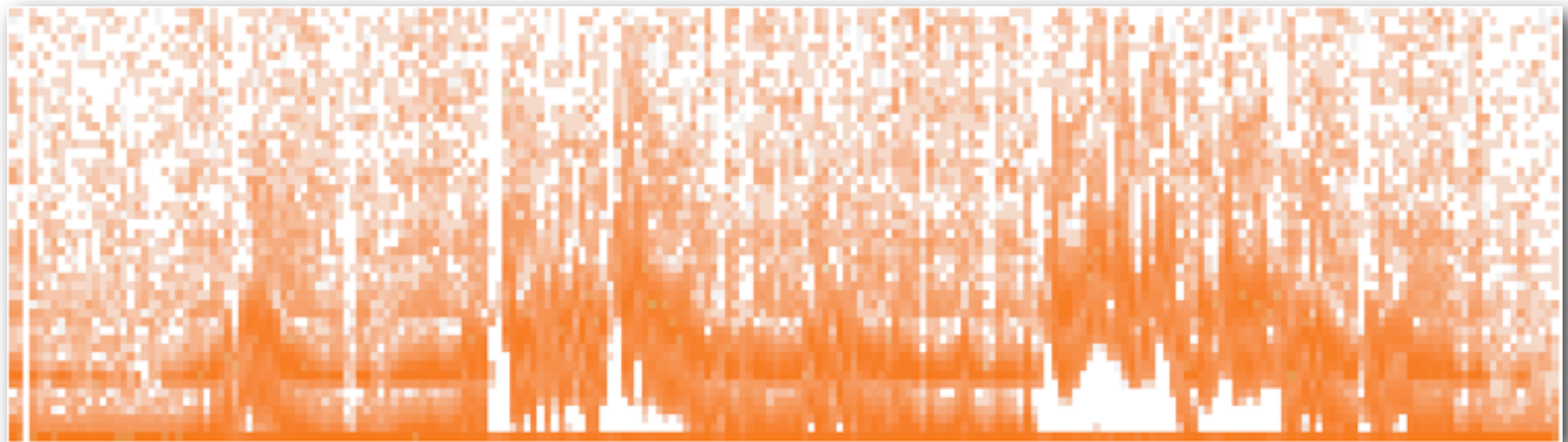
- Could visualize latency as a scalar (i.e., average):



- This hides outliers — and in a real-time system, it is the outliers that you care about!
- Using percentiles helps to convey distribution — but crucial detail remains hidden

# Visualizing latency as a heatmap

- Latency is much better visualized as a heatmap, with time on the x-axis, latency on the y-axis, and frequency represented with color saturation:

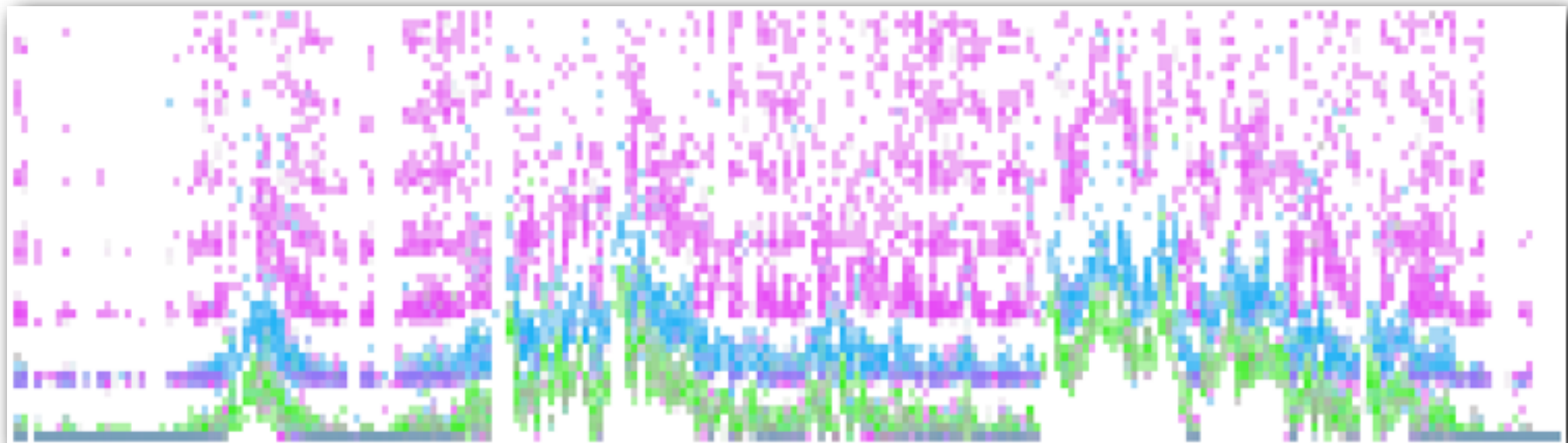


- Many patterns are now visible (as in this example of MySQL query latency), but critical data is still hidden



# Visualizing latency as a 4D heatmap

- Can use hue to represent higher dimensionality: time on the x-axis, latency on the y-axis, frequency via color saturation, and hue representing the new dimension:

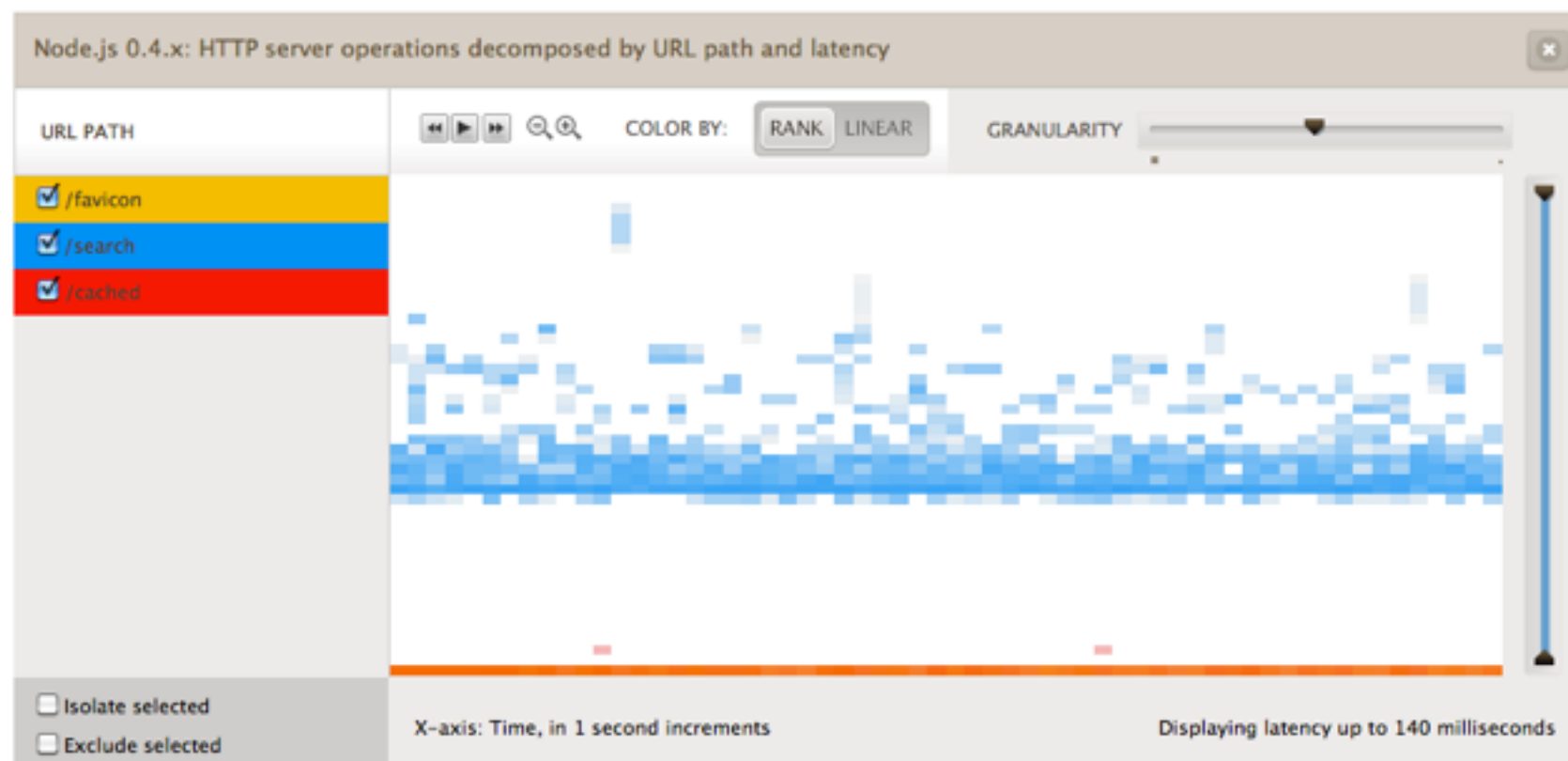


- In this example, the higher dimension is the MySQL database table associated with the operation

# Visualizing node.js latency



- Using the USDT probes as foundation, we developed a *cloud analytics* facility that visualizes latency in real-time via four dimensional heatmaps:



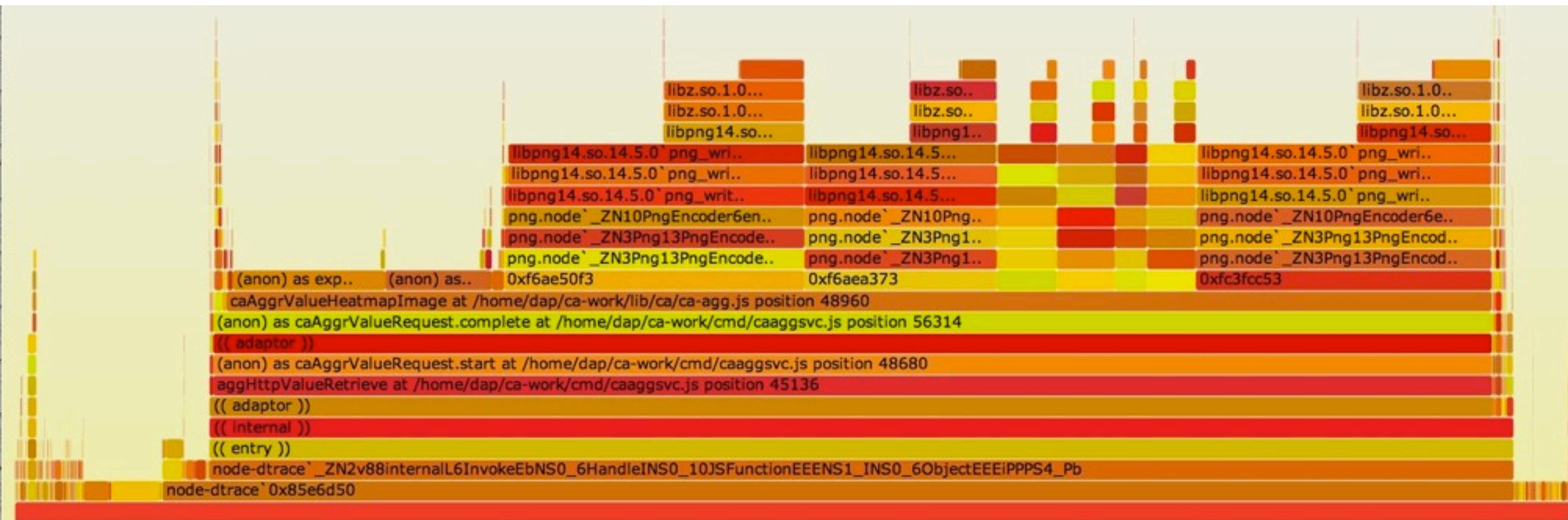
- Facility is available via Joyent's no.de service, Joyent's public cloud, or Joyent's SmartDataCenter

- Latency visualization is essential for understanding *where* latency is being induced in a complicated system, but how can we determine *why*?
- This requires associating an external event — an I/O request, a network packet, a profiling interrupt — with the code that's inducing it
- For node.js — like other dynamic environments — this is historically very difficult: the VM is opaque to the OS
- Using DTrace's helper mechanism, we have developed a *V8 ustack helper* that allows OS-level events to be correlated to the node.js-backtrace that induced them
- Available for node 0.6.7 on Joyent's SmartOS

# Visualizing node.js CPU latency



- Using the node.js ustack helper and the DTrace profile provider, we can determine the relative frequency of stack backtraces in terms of CPU consumption
- Stacks can be visualized with flame graphs, a stack visualization developed by Joyent's Brendan Gregg:



- node.js is particularly amenable for the DIRTy apps that typify the real-time web
- The ability to understand latency must be considered when deploying node.js-based systems into production!
- Understanding latency requires *dynamic instrumentation* and *novel visualization*
- At Joyent, we have added DTrace-based dynamic instrumentation for node.js to SmartOS, and novel visualization into our cloud and software offerings
- Better production support — better observability, better debuggability — remains an important area of node.js development!



- node.js is adept at *connecting* components in the system; it is unlikely to be the only component!
- As such, when using node.js to develop a DIRTy app, you can expect to spend as much time (if not more!) understanding the *components* as the app
- When selecting components — operating system, in-memory data store, database, distributed data store — *observability* must be a primary consideration!
- When building a team, look for *full-stack* engineers — DIRTy apps pose a full-stack challenge!

# Thank you!



- @ryah, @rmustacc and @dapsays for Node DTrace USDT integration
- @dapsays, @rmustacc, @rob\_ellis and @notmatt for cloud analytics
- @chrisandrews for node-dtrace-provider and @mcavage for putting it to such great use in ldap.js
- @dapsays for the V8 DTrace ustack helper
- @brendangregg for both the heatmap and flame graph visualizations — and for his excellent DTrace book!
- More information: <http://dtrace.org/blogs/dap>, <http://dtrace.org/blogs/brendan> and <http://smartos.org>

# ArchSummit

中国·深圳 2012.08

## INTERNATIONAL ARCHITECT SUMMIT

全球架构师峰会

详情请访问: [architectsummit.com](http://architectsummit.com)

• **3**天 • **6**场主题演讲

• **3**场圆桌论坛 • **9**场专题会议

• 国内外**30**余家IT、互联网公司的**50**多位来自一线的讲师齐聚一堂

主办方: **InfoQ**

战略合作伙伴: **Tencent 腾讯**

特别支持:



<http://architectsummit.com>





# QCon

杭州站 · 2012年10月25日~27日

[www.qconhangzhou.com](http://www.qconhangzhou.com) (6月启动)

QCon北京站官方网站和资料下载

[www.qconbeijing.com](http://www.qconbeijing.com)