On Observability

Richard Hartmann,
RichiH@{freenode,OFTC,IRCnet},
richih@{debian,fosdem,richih}.org,
@TwitchiH

2019-02-03

‘whoami‘

- Richard ”RichiH” Hartmann
- Swiss army chainsaw at SpaceNet
  - Leading the build of one of the most modern datacenters in Europe
  - ...and always looking for nice co-workers in the Munich area
- FOSDEM, DebConf, DENOGx, PromCon staff
- Author of `https://github.com/RichiH/vcsh`
- Debian Developer
- Prometheus team member
- OpenMetrics founder

Buzzword

buzzword, n:
A useful concept which has been picked up by everyone without understanding its deeper meaning and used so often that it's devoid of its original context and definition.
May revert to usefulness in the same or different meaning, or die off.

Cargo culting

cargo culting, v:
Villagers on remote Pacific islands observed U.S. soldiers building marker fires and runways during WWII; this made planes come and bring gifts from the heavens. Cults emerged which built bonfires and runways in the hopes of getting more gifts.
Also see: *copy & paste*

Monitoring

monitoring, n:
Old buzzword.
Too often: focus is put on collecting, persisting, and alerting on just any data, as long as its data.
It might also be garbage.
Also see: *data lake*

Observability

observability, n:
Function of a system with which humans and machines can observe, understand, and act on the state of said system.
Or: Being able to make deductions about the internal state of a system by looking at inputs and outputs only.

Thanks!

Thanks for listening!

Questions?

Email me if you want a job in Munich.

See slide footer for contact info.

Learnings

- Baseline of monitoring
- Types of monitoring data and when to use them
- Types of complexity
- Containing complexity
- Service, contracts, SL{I,O,A}, etc
- Services upon services
- Bringing it all together

Recap

Monitoring is the bedrock of everything (in IT).

Hope is not a strategy.

Claim

Uninformed, or cargo culted, monitoring equals hope.
Also see: *ISO 9001 & 27001*

So we need informed decisions, made on a factual basis.

50:50

Broadly speaking, there are metrics and events

Metrics: Development over time

Events: Specific points in time

Metrics

- Numerical data
  - Counters: Things going up monotonically, e.g. total transmitted bytes
  - Gauges: Things going up and down, e.g. temperatures
  - Bool/ENUM: Special case of gauges indicating a changing state or a singular event
  - Histograms and percentiles: Things going into buckets or being in a specific percentage band, e.g. latency
- Counters and histograms lose, or compress, data (in the common case)
- Easy to handle at scale
- You can do math on them!

Logs

- Most likely text items
- Usually with inlined metadata
- Scale linearly with service load
- Can be summarized into counters, histograms, and quantiles

Traces

- Execution path along the, hopefully annotated, code
- Impacts code runtime, aka expensive
- Can hide race conditions and other timing-dependent issues
- Usually disabled or sampled

Dumps

- Thrown when programs abort abnormally
- Execution path along the code
- Not annotated unless compiler artefacts of the exact same program are available
- You want to avoid them, but you also want to collect them when they happen

When to use what

- Metrics should usually be the first point of entry
  - ..for alerts
  - ..for dashboards
  - ..for data exploration
- Logs are usually the second step
  - ..for establishing order of events
  - ..for detailed information
  - ..for access control, due diligence, etc
- Traces and dumps are useful to understand why individual system components behave in a certain way

Types of complexity

Fake complexity, aka shitty design

System-inherent complexity

Handling complexity

You can reduce fake complexity

You can contain inherent complexity

Containing complexity

You need to compartmentalize complexity to make it manageable

What's a service?

A service is anything a different entity relies upon

This entity might be another team, a customer, or yourself

Handover

Service delineations have many names: interface, API, contract

I like to think of all of them as contracts. Why?

Tetris

Services build on top of each other

(Network * x + machine/container/kubelet * y + daemon/microservice * z) * n = HTTP service

Jenga

This tower can topple if the underlying building blocks are removed without due consideration.

”Contract” implies a firm commitment, which is why I like this term.

Chinese whispers

There's another common term for contract: layer.

Imagine if someone simply changed how IP works.

Trolling

For example, imagine someone would claim that IP addresses have 128 instead of 32 bits all of sudden...

Cake

So we agree that layering makes sense, but why do we agree?

It's complicated

Because we internalized that it's good practice to contain and compartmentalize system-inherent complexity.

Spectre, Meltdown, etc

A CPU is highly complex, but we are happy to trust its hidden complexity because there's a well-defined service boundary.

Relevance

Customers care about their services being up, not about individual service components

Discern between primary (service-relevant) and secondary (informational / debugging) SLIs; alert only on the former

Anything currently or imminently impacting customer service must be alerted upon.

Containment

Service delineations are the perfect boundaries for containing complexity

What's all this, now?

Monitoring tells you whether the system works.
Observability lets you ask why it's not working.

-

Baron Schwartz

Observability is not something you ever achieve, you can always improve on it.

As such there's not the One True Thing to do, it's about establishing the correct mindset.

BCPs

- Every outage gets a blame-free(!) post-mortem; and this includes a review of all relevant SLI & SLO
  - ..are they still useful?
  - ..would you have been quicker if you would have had different/more data?
  - ..should you retire some data collection?
- Link services together in your dashboards, etc
  - Make jumping into underlying services and their data as fluent as possible
  - Surface important insights from underlying services as context

BCPs

- Avoid relying only on blackbox data where possible; you need to get into your systems and extract fine-grained and meaningful data
  - Best case, this means instrumenting your code to extract metrics and traces
    Every time you are even considering to place a DEBUG statement into a codepath, put a counter
  - In the networking space, this often means requiring better data from your vendors.
    Explain what and why you need it, then force them via conditional POs, etc.

BCPs

- Collecting 10x or 100x more data means you have more substance to work with
- Avoid data lakes, attach meaningful metadata as early as possible
- Your tools must be able to handle this load
- Even more important, they must make handling the amounts of data manageable, and support automation

BCPs

- You (hopefully) know your services best, so create debugging stories in advance
  - Which critical paths do your users have, and where to you need to emit relevant signals?
  - What are common and/or critical paths while getting visibility into issues?
  - Which parts of these paths can you automate (more)?
  - Does it makes sense to introduce new (internal) service boundaries and/or contracts to create new compartments?
  - At what point will your users be happy again?
- ELI5: Find an non-expert and explain your services to them

Thanks!

Thanks for listening!

Questions?

Email me if you want a job in Munich.

See slide footer for contact info.