What does "Monitoring" mean?

Let's Stop Talking Past Each Other

Brian Brazil
Founder

Robust **Perception**

Who am I?

Engineer passionate about running software reliably in production.

- Core developer of Prometheus
- Studied Computer Science in Trinity College Dublin.
- Google SRE for 7 years, working on high-scale reliable systems.
- Contributor to many open source projects, including Ansible, Python, Aurora and Zookeeper.
- Founder of Robust Perception, provider of commercial support and consulting for Prometheus.

Monitoring

"Monitoring"

"Mon-i-tor-ing"

"Mon-i-tor-ing"

Do we all think of the same things when we see this word?

Historical Roots

A lot of what we do today for monitoring is based on tools and techniques that were awesome decades ago.

Machines and services were cared for by artisan sysadmins, with loving individual attention.

Special cases were the norm.

Historical Roots

Even though the tools have moved on greatly in both the open source and commercial spaces, practices haven't always moved as quickly.

Often still feeding the machine with human blood.

Let's look at some of the tools of where we came from.

A little history - MRTG and RRD

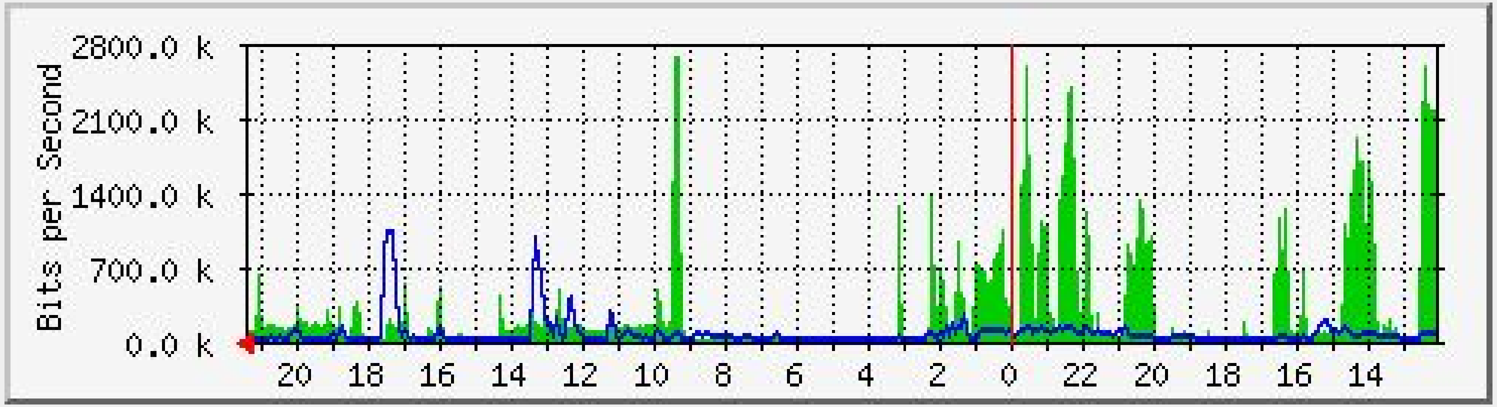
In 1994 Tobias Oetiker created a perl script, which became MRTG 1.0 in 1995. Used to graph metrics from SNMP or external programs. Stored data in constantly rewritten ASCII file.

MRTG 2.0 moved some code to C, released 1997.

RRD started in 1997 by Tobias Oetiker to further improve performance, released 1999.

Many tools use/used RRD, e.g. Graphite and Munin.

A little history - MRTG and RRD



Source: Wikipedia

Robust **Perception**

A little history - Nagios

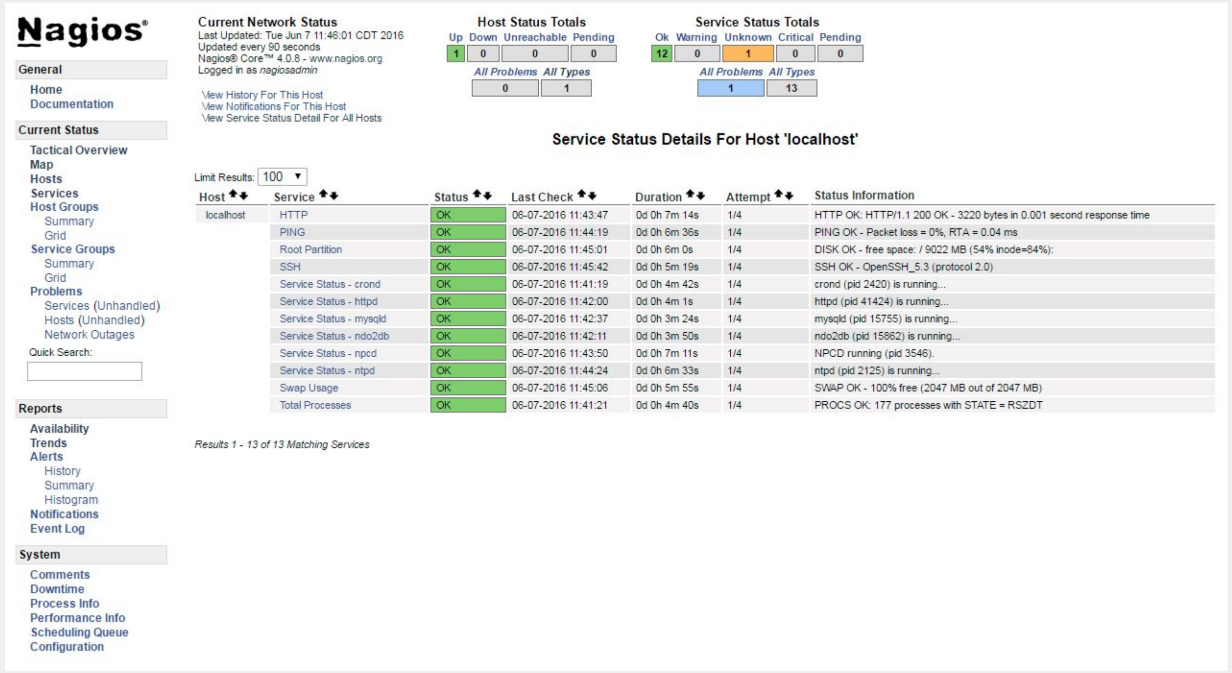
Written initially by Ethan Galstad in 1996. MS-DOS application to do pings.

Started more properly in 1998, first release in 1999 as NetSaint (renamed in 2002 for legal reasons).

Runs scripts on a regular basis, and sends alerts based on their exit code.

Many projects inspired by/based off Nagios such as Icinga, Sensu and Zmon.

A little history - Nagios



Current Network Status
Last Updated: Tue Jun 7 11:46:01 CDT 2016
Updated every 90 seconds
Nagios® Core™ 4.0.8 - www.nagios.org
Logged in as *nagiosadmin*

View History For This Host
View Notifications For This Host
View Service Status Detail For All Hosts

Host Status Totals

Up	Down	Unreachable	Pending
1	0	0	0

All Problems	*All Types*
0	1

Service Status Totals

Ok	Warning	Unknown	Critical	Pending
12	0	1	0	0

All Problems	*All Types*
1	13

Service Status Details For Host 'localhost'

Limit Results: 100

Host	Service	Status	Last Check	Duration	Attempt	Status Information
localhost	HTTP	OK	06-07-2016 11:43:47	0d 0h 7m 14s	1/4	HTTP OK: HTTP/1.1 200 OK - 3220 bytes in 0.001 second response time
	PING	OK	06-07-2016 11:44:19	0d 0h 6m 36s	1/4	PING OK - Packet loss = 0%, RTA = 0.04 ms
	Root Partition	OK	06-07-2016 11:45:01	0d 0h 6m 0s	1/4	DISK OK - free space: / 9022 MB (54% inode=84%):
	SSH	OK	06-07-2016 11:45:42	0d 0h 5m 19s	1/4	SSH OK - OpenSSH_5.3 (protocol 2.0)
	Service Status - crond	OK	06-07-2016 11:41:19	0d 0h 4m 42s	1/4	crond (pid 2420) is running...
	Service Status - httpd	OK	06-07-2016 11:42:00	0d 0h 4m 1s	1/4	httpd (pid 41424) is running...
	Service Status - mysqld	OK	06-07-2016 11:42:37	0d 0h 3m 24s	1/4	mysqld (pid 15755) is running...
	Service Status - ndo2db	OK	06-07-2016 11:42:11	0d 0h 3m 50s	1/4	ndo2db (pid 15862) is running...
	Service Status - npcd	OK	06-07-2016 11:43:50	0d 0h 7m 11s	1/4	NPCD running (pid 3546).
	Service Status - ntpd	OK	06-07-2016 11:44:24	0d 0h 6m 33s	1/4	ntpd (pid 2125) is running...
	Swap Usage	OK	06-07-2016 11:45:06	0d 0h 5m 55s	1/4	SWAP OK - 100% free (2047 MB out of 2047 MB)
	Total Processes	OK	06-07-2016 11:41:21	0d 0h 4m 40s	1/4	PROCS OK: 177 processes with STATE = RSZDT

Results 1 - 13 of 13 Matching Services

Source: Wikipedia

Robust **Perception**

Historical Heritage

These very popular tools and their offspring left us with a world where graphing is whitebox and alerting is blackbox, and they are separate concerns.

They come from a world where machines are pets, and services tend to live on one machine.

They come from a world where even slight deviance would be immediately jumped upon by heroic engineers in a NOC.

We need a new perspective in a cloud native environment.

So what is monitoring?

We need a view that goes beyond alerting, graphing and jumping on every potential problem.

We need to consider additional data sources, such as logs and browser events.

What about looking at the problem statement rather than what the tools can give us?

What is the ultimate goal of all this "monitoring"?

Why do we monitor?

- Know when things go wrong
- Be able to debug and gain insight
- Trending to see changes over time
- Plumbing data to other systems/processes

Knowing when things go wrong

The first thing many people think of you say monitoring is alerting.

What is the wrongness we want to detect and alert on?

A blip with no real consequence, or a latency issue affecting users?

Symptoms vs Causes

Humans are limited in what they can handle.

If you alert on every single thing that might be a problem, you'll get overwhelmed and suffer from alert fatigue.

Key problem: You care about things like user facing latency. There are hundreds of things that could cause that.

Alerting on every possible cause is a Sisyphean task, but alerting on the symptom of high latency is just one alert.

Example: CPU usage

Some monitoring systems don't allow you to alert on the latency of your servers.

The closest you can get is CPU usage.

False positives due to e.g. logrotate running too long.

False negatives due to deadlocks.

End result: Spammy alerts which operators learn to ignore, missing real problems.

Human attention is limited

Alerts should require intelligent human action!

Alerts should relate to actual end-user problems!

Your users don't care if a machine has a load of 4.

They care if they can't view their cat videos.

Debugging to Gain Insight

After you receive an alert notification you need to investigate it.

How do you work from a high level symptom alert such as increased latency?

You methodically drill down through your stack with dashboards to find the subsystem that's the cause.

Break out more tools as you drill down into the suspect process.

Complementary Debugging Tools

Trending and Reporting

Alerting and debugging is short term.

Trending is medium to long term.

How is cache hit rate changing over time?

Is anyone still using that obscure feature?

When will I need more machines?

Plumbing

When all your have is a hammer, everything starts to look like a nail.

Often it'd be really convenient to use a monitoring system as a data transport as part of some other process (often a control loop of some form).

This isn't monitoring, but it's going to happen.

If it's ~free, it's not necessarily even a bad idea.

How do we monitor?

Now knowing the three general goals of monitoring (plus plumbing), how do we go about it?

What data do we collect? How do we process it?

What tradeoffs do we make?

Monitoring resources aren't free, so everything all the time is rarely an option.

The core is the Event

Events are what monitoring systems work off.

An event might be a HTTP request coming in, a packet being sent out, a library call being made or a blackbox probe failing.

Events have context, such as the customer ultimately making the request, which machines are involved, and how much data is being processed.

A single event might accumulate thousands of pieces of context as it goes through the system, and there can be millions of events per second.

How do we monitor?

We're going to look at 4 classes of approaches to monitoring.

- Profiling
- Metrics
- Logs
- Distributed Tracing

When someone says "monitoring" they often have one of these stuck in their head, however they're all complementary with different tradeoffs.

Profiling

Profiling is using tools like tcpdump, gdb, strace, dtrace, BPF and pretty much anything Brendan Gregg talks about.

They give you very detailed information about individual events.

So detailed that you can't keep it all, so use must be targeted and temporary.

Great for debugging if have an idea what's wrong, not so much for anything else.

Metrics

Metrics make the tradeoff that we're going to ignore individual events, but track how often particular contexts show up. Examples: Prometheus, Graphite.

For example you wouldn't track the exact time each HTTP request came in, but you do track enough to tell that there were 3594 in the past minute.

And 14 of them failed. And 85 were for the login subsystem. And 5.4MB was transferred. With 2023 cache hits.

And one of those requests hit a weird code path everyone has forgotten about.

Metric Tradeoffs

Metrics give you breadth.

You can easily track tens of thousands of metrics, but you can't break them out by too many dimensions. E.g. breaking out metrics by email address is rarely a good idea.

Great for figuring out what's generally going on, writing meaningful alerts, narrowing down the scope of what needs debugging.

Not so great for tracking individual events.

Logs

Logs take the opposite approach to metrics.

They track individual events. So you can tell that Mr. Foo visited /myendpoint yesterday at 7pm and received a 7892 byte response with status code 200.

The downside is you're limited in how many fields you can track due, likely <100.

Data volumes involved can also mean it takes some time for data to be available.

Examples: ELK stack, Graylog

Logs - But Wait There's More

Just like with how "monitoring" tends to have per-person pre-conceived ideas there's a similar issue with "logs". Can lead to people talking past each other.

Roughly four types of logs:

- Transactional, e.g. for billing or auditing. Cannot be lost.
- Request, e.g. logs for individual HTTP requests or failures.
- Application, e.g. I'm starting GC now, I got a SIGHUP.
- Debug, e.g. individual function calls. Essentially profiling.

Each has different volume, ease of processing and durability requirements.

Distributed Tracing

Distributed tracing is really a special case of logging.

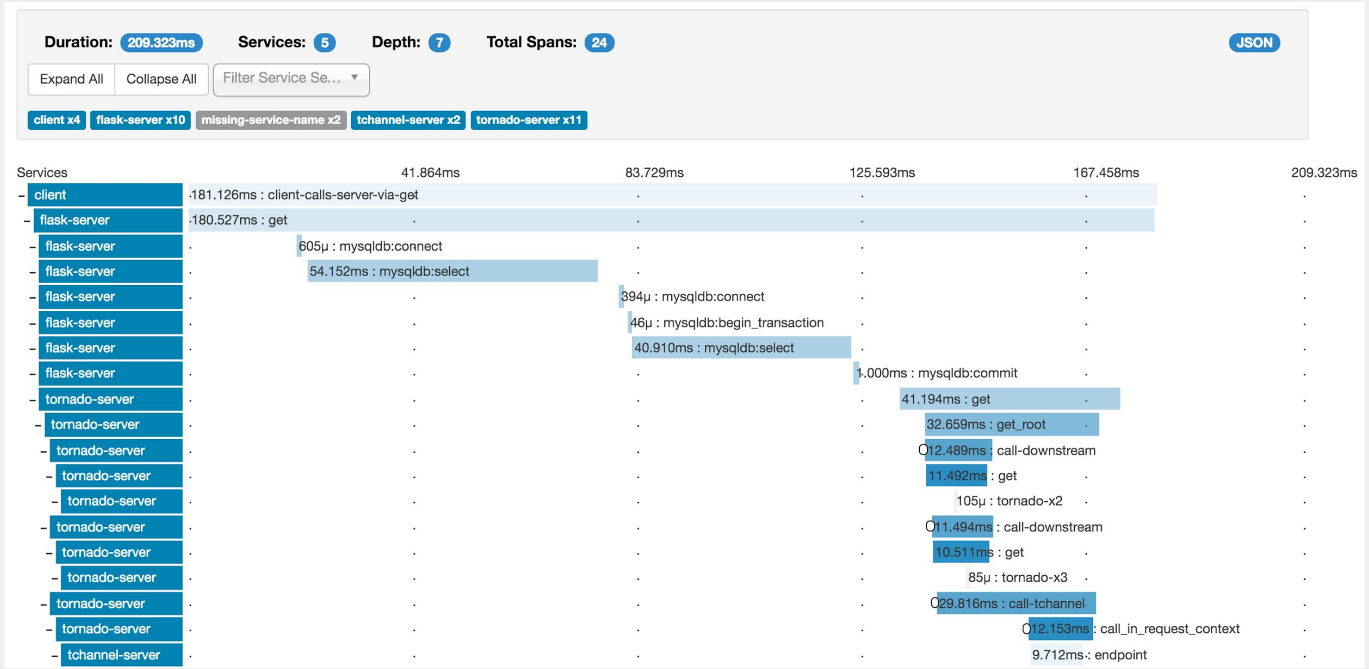
It gives each request an unique ID that is logged as the request is handled through various services in your architecture.

It ties these logs back together to see how the request flowed through the system, and where time was spent.

Essential for debugging large-scale distributed systems.

Examples: OpenTracing, OpenZipkin

Distributed Tracing


Duration: 209.323ms **Services:** 5 **Depth:** 7 **Total Spans:** 24 JSON

Expand All | Collapse All | Filter Service Se...

client x4 | flask-server x10 | missing-service-name x2 | tchannel-server x2 | tornado-server x11

Services · 41.864ms · 83.729ms · 125.593ms · 167.458ms · 209.323ms

Services	Span
client	181.126ms : client-calls-server-via-get
flask-server	180.527ms : get
flask-server	605μ : mysqldb:connect
flask-server	54.152ms : mysqldb:select
flask-server	394μ : mysqldb:connect
flask-server	46μ : mysqldb:begin_transaction
flask-server	40.910ms : mysqldb:select
flask-server	1.000ms : mysqldb:commit
tornado-server	41.194ms : get
tornado-server	32.659ms : get_root
tornado-server	12.489ms : call-downstream
tornado-server	11.492ms : get
tornado-server	105μ : tornado-x2
tornado-server	11.494ms : call-downstream
tornado-server	10.511ms : get
tornado-server	85μ : tornado-x3
tornado-server	29.816ms : call-tchannel
tornado-server	12.153ms : call_in_request_context
tchannel-server	9.712ms : endpoint

Source: OpenZipkin

So what does "monitoring" mean?

Not just blackbox alerts going to a NOC

Not just per-host graphs

Not just aggregated whitebox application graphs

Not just logs

Not just metrics

So what does "monitoring" mean?

Monitoring is the set of tools and techniques you use to keep an eye on how your system is doing, and keep it functional.

There is no silver bullet product you can get that does everything.

Culture and policy form part of your monitoring system too!

In Summary: Monitoring Goals

- Know when things go wrong
  - Have alerts that require intelligent human action and directly relate to user impact
- Be able to debug and gain insight
  - Be able to methodically go from an alert to the culprit subsystem
- Trending to see changes over time
  - Engineering depends on data
- Plumbing data to other systems/processes
  - It's going to happen

In Summary: Monitoring Systems

- Profiling
  - Lots of data - too much to use it all the time.
- Metrics
  - Great breadth - at the cost of depth.
- Logs
  - Great depth - at the cost of breadth.
- Distributed Tracing
  - Essential to debugging certain distributed systems problems.

Final Word

Don't be held back by what made sense 20 years ago, or what you currently do.

As we go towards a cloud native world we need to take advantage of all the new classes of monitoring tools out there. We must scale ourselves as engineers.

Just as we no longer provision machines individually, neither should we care about once machine being slow/down/dodgy. Care about services and end-users.

(Also, I hear Prometheus is great for metrics)

Robust **Perception**

Questions?

No really, you should try Prometheus: prometheus.io

Here's a Demo: demo.robustperception.io

My Company Website: www.robustperception.io

Robust **Perception**