The ideas in this paper form the foundation for
Monitoring distributed systems: A case study in how Google monitors its complex systems,
Which is a chapter in Site Reliability Engineering: How Google Runs Production Systems;
those versions are more detailed and more carefully edited for tybograbhical emors.

For some other discussion, see the Hacker News post from mid-2014.

My Philosophy on Alerting

based my observations while I was a Site Reliability Engineer at Google

Author: Rob Ewaschuk <rob@infinitepigeons.org>

Introduction
Vernacular
Monitor for your users
Cause-based alerts are bad (but sometimes necessary)
    Alerting from the spout (or beyond!)
    Causes are still useful
Tickets, Reports and Email
Playbooks
Tracking & Accountability
You're being naïve!
Summary

Summary

When you are auditing or writing alerting rules, consider these things to keep your oncall rotation happier:

- Pages should be urgent, important, actionable, and real.
- They should represent either ongoing or imminent problems with your service.
- Err on the side of removing noisy alerts – over-monitoring is a harder problem to solve than under-monitoring.
- You should almost always be able to classify the problem into one of: availability & basic functionality; latency; correctness (completeness, freshness and durability of data); and feature-specific problems.
- Symptoms are a better way to capture more problems more comprehensively and robustly with less effort.
- Include cause-based information in symptom-based pages or on dashboards, but avoid alerting directly on causes.
- The further up your serving stack you go, the more distinct problems you catch in a single rule. But don't go so far you can't sufficiently distinguish what's going on.

The ideas in this paper form the foundation for
Monitoring distributed systems: A case study in how Google monitors its complex systems,
Which is a chapter in Site Reliability Engineering: How Google Runs Production Systems;
those versions are more detailed and more carefully edited for tybograbhical emors.

For some other discussion, see the Hacker News post from mid-2014.

- If you want a quiet oncall rotation, it's imperative to have a system for dealing with things that need timely response, but are not imminently critical.

Introduction

After seven years of being oncall for a variety of different services, including both massive- and small-scale, fast-moving products, and several parts of core infrastructure, I have developed a philosophy on monitoring and alerting. It reflects my fundamental view on pages and pagers:

- Every time my pager goes off, I should be able to **react with a sense of urgency**. I can only do this a few times a day before I get fatigued.
- Every page should be **actionable**; simply noting "this paged again" is not an action.
- Every page should **require intelligence** to deal with: no robotic, scriptable responses.

Overall, it's a bit aspirational, but it guided me when I wrote or reviewed a new paging rule in the monitoring systems. These are some questions I use when I'm writing or reviewing a new rule that might result in a page:

- Does it detect an **otherwise undetected condition** that is urgent, actionable and actively or imminently user-visible. Note that "N+0" zero-redundancy situations count as imminent, as do "nearly full and getting fuller" parts of your service, like storage being maxed out.
- Will I ever **be able to ignore this rule**, knowing it's benign? When and why, and can I refine the rule to avoid this situation?
- Is it identifying a situation which is **definitely (going to be) hurting users**? Are there detectable cases where it doesn't hurt users that should be filtered out? Think things like server clusters with test-traffic, etc.
- Can I **take action** in response to this alert? Is that action urgent, or could it wait until after I wake up or the end of the weekend or next quarter?
- Are other people getting paged at the same time? Are they going to fix the problem? Or maybe I'm going to fix the problem for someone else? Can we tie these things together? Can my version of the rule wait a bit for them to try to fix it?

The ideas below are certainly aspirational—no pager rotation of a growing, changing service is ever as clean as it could be—but there are some tricks that get you much closer.

The ideas in this paper form the foundation for
Monitoring distributed systems: A case study in how Google monitors its complex systems,
Which is a chapter in Site Reliability Engineering: How Google Runs Production Systems;
those versions are more detailed and more carefully edited for tybograbhical emors.

For some other discussion, see the Hacker News post from mid-2014.

Vernacular

This document uses the following terms:

- **page**: anything that tries to urgently and actively get the attention of a specific human (e.g. via a pager or cell phone going beep beep beep)
- **rule**: any kind of logic for detecting some interesting condition in any monitoring system.
- **alert**: is a manifestation of a rule that (intends to) reach a human, e.g. as a page, an email, a message in an IRC channel, auto-filing a ticket, etc.

Monitor for your users

I call this "**symptom-based monitoring**," in contrast to "cause-based monitoring". Do your users care if your MySQL servers are down? No, they care if their queries are failing. (Perhaps you're cringing already, in love with your Nagios rules for MySQL servers? Your users don't even know your MySQL servers *exist*!) Do your users care if a support (i.e. non-serving-path) binary is in a restart-loop? No, they care if their features are failing. Do they care if your data push is failing? No, they care about whether their results are fresh.

Users, in general, care about a small number of things:

- **Basic availability and correctness**. no "Oops!", no 500s, no hung requests or half-loaded pages or missing Javascript or CSS or images or videos. Anything that breaks the core service in some way should be considered unavailability.
- **Latency.** Fast. Fast. Fast. Also, fast.
- **Completeness/freshness/durability.** Your users data should be safe, should come back when you ask, and search indices should be up-to-date. Even if it is temporarily unavailable, users should have complete faith that it's coming back unc#rÃ÷ʊted.
- **Features.** Your users care that all the features of the service work—you should be monitoring for anything that is an important aspect of your service even if it's not core functionality/availability (e.g. the Calculator and stock ticker showing up in search results).

That's pretty much it. There's a subtle but important difference between *database servers* being unavailable and *user data* being unavailable. The former is a proximate cause, the latter is a symptom. You can't always cleanly distinguish these things, particularly when you don't have a

The ideas in this paper form the foundation for
Monitoring distributed systems: A case study in how Google monitors its complex systems,
Which is a chapter in Site Reliability Engineering: How Google Runs Production Systems;
those versions are more detailed and more carefully edited for tybograbhical emors.

For some other discussion, see the Hacker News post from mid-2014.

way to mimic the client's perspective (e.g. a blackbox probe or monitoring their perspective directly). But when you can, you should.

Cause-based alerts are bad (but sometimes necessary)

"But," you might say, "I *know* database servers that are unreachable results in user data unavailability." That's fine. *Alert on the data unavailability.* Alert on the symptom: the 500, the Oops!, the whitebox metric that indicates that not all servers were reached from the database's client. Why?

- **You're going to have to catch the symptom *anyway*.** Maybe it can happen because of network disconnection, or CPU contention, or myriad other problems you haven't thought of yet. So you have to catch the symptom.
- Once you catch the symptom and the cause, you have **redundant alerts**; these need separate tuning, and result in either duplication or complicated dependency trees
- **The allegedly inevitable result is not always inevitable**: maybe your database servers are unavailable because you're turning up a new instance or turning down an old one. Or maybe a feature was added to do fast-failover of requests, and so you don't care anymore about a single server's availability. Sure, you can catch all these cases with increasingly complicated rules, but why bother? The failure mode is more bogus pages, more confusion, and more tuning, with no gain, and *less time spent on fixing the alerts that matter*.

But sometimes they're necessary. There's (often) no symptoms to "almost" running out of quota or memory or disk I/O, etc., so you want rules to know you're walking towards a cliff. **Use these sparingly; don't write cause-based paging rules for symptoms you can catch otherwise.**

Alerting from the spout (or beyond!)

The best alerts in a layered client/server system come from the client's perspective:

- The **client sees the results of retries, network latency between client & server**, and has a better perspective on the user-facing latency and errors than the server
- In many cases the client (e.g. a mixer or application server) is **aggregating responses from many backends**, like caching services, databases, account management/authorization services, query shards, etc. Your monitoring is more robust to changes in underlying infrastructure (and in application-level failover and retries) if you **see what the client *actually* does**.

The ideas in this paper form the foundation for
Monitoring distributed systems: A case study in how Google monitors its complex systems,
Which is a chapter in Site Reliability Engineering: How Google Runs Production Systems;
those versions are more detailed and more carefully edited for tybograbhical emors.

For some other discussion, see the Hacker News post from mid-2014.

- In many cases, the client can present a simpler view of the world than the backends. For example if a request fans out to hundreds of query-servers, each query server has too limited of a view of the world to be a useful source of alerting.

For many services, this means alerting on what your front-most load-balancers see in terms of latency, errors, etc. This way you only see the result of broken servers *if those results are making it to the user*. Conversely, you're seeing a bigger class of problems than you can see from your servers: if they're all down, or serving out uncounted 500s, or dropping 10% of connections on the floor, your load balancer knows but your server might not.

Note that going too far can introduce agents that are beyond your control and responsibility. If you can reliably capture a view of exactly what your users sees (e.g. via browser-side instrumentation), that's great! But remember that signal is full of noise—their ISP, browser, client-side load and performance—so it probably shouldn't be the only way you see the world. It may also be lossy, if your external monitoring can't always contact you. Taken to this kind of extreme, it's still a useful signal but maybe not one you want to page on.

Causes are still useful

Cause-based rules can still be useful. In particular, they can help you jump quickly to a known deficiency in your production system.

If you gain a lot of value in automatically tying symptoms back to causes, perhaps because there are causes that are outside of your control to eliminate, I advocate this technique:

1. When you write (or discover) a rule that represents a cause, check that the symptom is also caught. If not, make it so.
2. Print a terse summary of *all* of your cause-based rules that are firing in *every* page that you send out. A quick skim by a human can identify whether the symptom they just got paged for has an already-identified cause. This might look like:

```
TooMany500StatusCodes
Served 10.7% 5xx results in the last 3 minutes!
Also firing:
      JanitorProcessNotKeepingUp
      UserDatabaseShardDown
      FreshnessIndexBehind
```

   In this case it's clear that the most likely source of 500s is a database problem; if instead the firing symptom had been that a disk was getting full, or that result pages were

The ideas in this paper form the foundation for
Monitoring distributed systems: A case study in how Google monitors its complex systems,
Which is a chapter in Site Reliability Engineering: How Google Runs Production Systems;
those versions are more detailed and more carefully edited for tybograbhical emors.

For some other discussion, see the Hacker News post from mid-2014.

coming back empty or stale, the other two causes might have been interesting.

3. Remove or tune cause-based rules that are noisy or persistent or otherwise low-value.

Using this approach, the mental burden of the mistuned, noisy rules has been changed from a pager beep & ack (and investigation, and followup, and..) to a single line of text to be skimmed over. Finally, since you need clear debugging dashboards anyway (for problems that don't start with an alert), this is another good place to expose cause-based rules.

That said, if your debugging dashboards let you move quickly enough from symptom to cause to amelioration, you don't need to spend time on cause-based rules anyway.

Tickets, Reports and Email

One way or another, you have some alerts that need attention soon, but not *right now*. I call these "sub-critical alerts".

- **Bug or ticket-tracking systems can be useful.** Having alerts open a bug can work out great, as long as multiple firings of the same alert get correctly threaded into a single ticket/bug. This system fails if there's no accountability for triaging and closing bugs; if the alert-opened bugs might go unseen for weeks, this clearly fails as a way of dealing with sub-critical alerts before they become critical! It also fails if your team is simply overloaded or is not assigning enough people to deal with followup; you need to be honest about how much time this is consuming, or you'll fall further and further behind.
- **A daily (or more frequent) report can work too.** One way this can work is to write sub-critical rules that are long-lived (e.g. "the database is over 90% full" or "we've served over 1000 very slow requests in the last day"), and send out a report periodically that shows all currently-firing rules. Again, without a system of accountability this amounts to less-spammy email alerts, so make sure the oncall person (or someone else) is designated to triage these every day (or every shift hand-off, or whatever works).
- **Every alert should be tracked through a workflow system.** Don't *only* dump them into an email list or IRC channel**.** In general, this quickly turns into specialized "foo-alerts" mailing lists or channels so that they can be summarily ignored. Except as a brief (usually days, at most weeks) period to vet that a new rule will not page too often, it's almost always a bad idea. It's also easy to ignore the volume of these alerts, and suddenly some old, mis-tuned rule is firing every minute for all of your thousand application servers, clogging up mailboxes. Oops.

The ideas in this paper form the foundation for
Monitoring distributed systems: A case study in how Google monitors its complex systems,
Which is a chapter in Site Reliability Engineering: How Google Runs Production Systems;
those versions are more detailed and more carefully edited for tybograbhical emors.

For some other discussion, see the Hacker News post from mid-2014.

The underlying point is to create a system that still has **accountability for responsiveness**, but doesn't have the **high cost of waking someone up**, interrupting their dinner, or preventing snuggling with a significant other.

Playbooks

Playbooks (or runbooks) are an important part of an alerting system; it's best to have an entry for each alert or family of alerts that catch a symptom, which can further explain what the alert means and how it might be addressed.

In general, if your playbook has a long detailed flow chart, you're **potentially spending too much time documenting what could be wrong and too little time fixing it**—unless the root causes are completely out of your control or fundamentally require human intervention (like calling a vendor). The best playbooks I've seen have a few notes about exactly what the alert *means*, and what's *currently* interesting about an alert ("We've had a spate of power outages from our widgets from VendorX; if you find this, please add it to Bug 12345 where we're tracking things for patterns".) Most such notes should be ephemeral, so a wiki or similar is a great tool.

Tracking & Accountability

Track your pages, and all your other alerts. If a page is firing and people just say "I looked, nothing was wrong", that's a pretty strong sign that you need to remove the paging rule, or demote it or collect data in some other way. Alerts that are less than 50% accurate are broken; even those that are false positives 10% of the time merit more consideration.

Having a system in place (e.g. a **weekly review of all pages, and quarterly statistics**) can help keep a handle on the big picture of what's going on, and tease out patterns that are lost when the pager is handed from one human to the next.

You're being naïve!

Yup, though I prefer the term "aspirational". Here are some great reasons to break the above guidelines:

- **You have a known cause that actually sits below the noise in your symptoms.** For example, if your service has 99.99% availability, but you have a common event that causes 0.001% of requests to fail, you can't alert on it as a symptom (because it's in the

The ideas in this paper form the foundation for
Monitoring distributed systems: A case study in how Google monitors its complex systems,
Which is a chapter in Site Reliability Engineering: How Google Runs Production Systems;
those versions are more detailed and more carefully edited for tybograbhical emors.

For some other discussion, see the Hacker News post from mid-2014.

noise) but you can catch the causing event. It might be worth trying to trickle this information up the stack, but maybe it really is simplest just to alert on the cause. *Caveat oncaller*.

- **You can't monitor at the spout, because you lose data resolution.** For example, maybe you tolerate some handlers/endpoints/backends/URLs being pretty slow (like a credit card validation compared to browsing items for sale) or low-availability (like a background refresh of an inbox). At your load balancers, this distinction may be lost. Walk down the stack and alert from the highest place where you have the distinction.
- **Your symptoms don't appear until it's too late, like you've run out of quota.** Of course, you need to page before it's too late, and sometimes that means finding a cause to page on (e.g. usage > 80% and will run out in < 4h at the growth rate of the last 1h). But if you can do that, you should also be able to find a similar cause that's less urgent (e.g. quota > 90% and will run out in < 4d at the growth rate of the last 1d) that will catch most cases, and deal with that as a ticket or email alert or daily problem report, rather than the last-ditch escalation that a page represents.
- **Your alert setup sound more complex than the problems they're trying to detect.** Sometimes they will be. The goal should be to tend towards simplicity, robust, self-protecting systems (how did you not notice that you were running out of quota? Why can't that data go somewhere else?) In the long term, they should trend towards simplicity, but at any given time the local optimum may be relatively complex rules to keep things quiet and accurate.

May the queries flow, and your pagers be quiet.