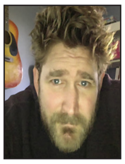
iVoyeur

Pager Trauma Statistics Daemon, PTSD

DAVE JOSEPHSEN



Dave Josephsen is the sometime book-authoring developer evangelist at Librato .com. His continuing mission: to help engineers worldwide close the feedback loop. dave-usenix@skeptech.org

I began writing this article late in the third and final day at Monitorama, the single-track monitoring conference. In fact, as I wrote these first few paragraphs, a man named Joey Parsons from Airbnb was talking about the health and well-being of Airbnb's on-call workers. In fact, "people matter" was something of a theme this year—a pretty drastic change for a normally tools and techniques focused crowd. It is certainly true, for whatever reason, that this Monitorama took a heavy turn toward introspection about measurement culture and especially on-call pain.

Over the last several years, a few speakers have talked about what can only be described as post-traumatic reactions to other people's ringtones, wherein they'll react in visceral, lizard-brain terror to the sound of someone else's phone, or even just ambient noise that sounds similar to their own phone's notification tone. This resonates very strongly with me, and I suspect that if you're reading this, it might resonate with you as well.

In 2012 I took a much needed break from Operations to travel the conference circuit, speaking and writing in the name of developer advocacy, outreach, and evangelism. By the time I finally quit Ops to find a job without an on-call component, I'd been on-call for seven years. Yep, every day and every night, as the (usually) sole technical lead for one startup or another.

I think I still underestimate the stress I put on myself and my family during this nearly decade-long on-call stint. I spent three consecutive Thanksgiving dinners alone at a datacenter in what became a running joke at my last startup: the "Thanksgiving Day Curse." The last time this happened in 2012, when my phone went off shortly after I sat down for Thanksgiving dinner, my exasperated wife dumped the contents of my plate into a large zip-lock bag and handed it to me on the way out the door.

Talks on "alert fatigue" have popped up now and again. I've even given one myself, but these normally focus on tools and techniques to reduce and filter false-positives. It seems to me that we rarely talk about the more intangible "people problems" that on-call work can introduce into our lives, or attempt to quantify the amount of pain on-call participants experience on a weekly basis. There are what can only be described as support groups, like the "I made a huge mistake" BoF Shawn Sterling has thrown for years at LISA, but these have always been tertiary events, rather than main-event fodder.

But because we are engineers, and especially because we are engineers who enjoy the measuring and quantification of things, many of the talks at Monitorama this year went beyond descriptions of the problem and described efforts to actually assess the cognitive and emotional stress that the computational first-responders in their organizations have to put up with. Many shared data, either in the form of graphs or spreadsheets, and a few even shared HR techniques like sleep-tracking, and mandatory three-day weekends for individuals coming off their on-call rotation.

I especially found the numbers fascinating. In Joey Parsons' talk "Monitoring and Health at Airbnb" [1], he shared that they've had some success at Airbnb making on-call a voluntary endeavor that is spread among engineers across every internal discipline. Airbnb's 50(!) on-call volunteers take three-day shifts and participate in weekly sysop meetings to keep everyone up to date with respect to chronic, ongoing production issues. The top on-call volunteer for the week before Monitorama fielded 48 production issues.

One of the most important things enabling teams like Airbnb to quantify their on-call stress is the use of third-party alert processing systems like PagerDuty and VictorOps. These systems export APIs that expose a wealth of information about every incident. Who was on-call when it happened, to whom it was assigned, the number of actual notifications that each incident spawned and on and on.

As I write this, I'm also in the process of transitioning back to Ops, so as you can imagine I'm certainly interested in seeing how our own on-call endeavors line up. We have, of course, no hope of finding 50 on-call volunteers in a company of < 20 people, but the ebb and flow of our production issues is a line I'd like to see. To that end, on the plane ride home I spent some time playing with the PagerDuty API, and landed with a rough first-pass at a Go program that can interrogate the PagerDuty API for total incidents, per-user pager notifications, and per-user acknowledgments, and forward these as counter increments to StatsD for visualization.

For the moment I'm calling it PTSD (Pager Trauma Statistics Daemon), and you can find it at http://github.com/djosephsen/ptsd. My hope is that it'll make it trivial for anyone using PagerDuty to get some stats about their on-call volume. It dumps these stats to STDOUT and a locally running StatsD instance.

Using Go

Increasingly, I find myself reaching for Go over something like Bash or Python to write API glue code like this. And I'd like to share the reasons with you, since I feel like it's rare to see real talk about Go in a context other than Web services and hardcore systems tooling. Here, I want to write some longish-term maintainable glue code to tie together a REST API and a service listening on a network socket.

Traditionally, Go has not been my first choice for parsing JSON responses from Web APIs. The built-in net/http library is fantastic for running queries, but the built-in JSON library, encoding/json, is a complete pain when it comes to working with large, complex, or unknown structures. In these cases I almost always find myself decoding into a hash-map (map[string]interface), and then manually type-casting my way to the real values. This process is error-prone, painful, and fragile, and you wind up with line upon line of incomprehensible nested expressions like:

```
'foo['people'].(map[string]interface)['dave'].(string)'
```

By comparison Bash and Python have lovely JSON parsing capabilities, but I've never really cared for their HTTP clients, which are sufficiently complicated that I've never really been able to commit their syntax to memory. Python, especially, is somewhat of a mess in this regard, with `urllib` vs. `requests` vs. `httplib`. To be clear, I'm a pretty down to earth practitioner who doesn't have anything in particular against any of these libraries, but it's basically a nightmare to have to go back and forth between them depending on the project/people I'm working on/with. I've seriously considered picking one and just severing ties with all of the people who use the others.

Recently, however, I discovered Anton Holmquist's "Jason" library [3], and I have to say it's made Go my...well, go-to language for working with JSON Web-APIs. Here's the pattern for making a GET request to an API and parsing the JSON response into something we can work with:

```
client := &http.Client{}
req, _ := http.NewRequest("GET", url, nil)
req.Header.Add("Authorization", authToken)
resp,err := client.Do(req)
body,err := jq.NewObjectFromReader(resp.Body)
```

First, we make a new `http.Client` object. This is unnecessary unless you need to do advanced things to the request like add headers. You can make simple requests directly with `http.Post()`. In this case, I add a header to submit my auth token. Once the request is built, we send it with `client.Do()`, and then we parse the response body directly into a JSON object with `NewObjectFromReader`, which we've imported from `antonholmquist/jason`.

Once the response is in a `json.Object`, a vast and powerful assortment of functions enable us to read out top-level or nested values straight into strings, ints, and etc. without any type-casting on our part. For example, to parse out a top-level `type` attribute into a Go-native string variable we would:

```
type_of_thingie := body.GetString("type")
```

There are even functions that return '[]*jason.Object,' which makes it completely trivial to iterate through arrays nested inside the JSON.

```
logs, _ := body.GetObjectArray("log_entries")
for _,log := range logs{
 fmt.Printf("log type :: %s", log.getString( "type"))
}
```

Needless to say, if I ever meet Anton Holmquist and he needs anything, like a ride home or a free beer or a dude with a truck to help him move, I'll happily oblige him. He's made my life much easier.

The second reason I chose Go was because I was going to want this to be a somewhat extensible project, by which I mean, I wanted to make it as easy as possible for other people to extend it without having to refactor. That meant implementing both the "collector" piece (e.g., PagerDuty) as well as the "outputter" piece (e.g., StatsD) as modules from the…well, get-go.

There are two reasons I find Go pretty nice for writing modular code. The first of these is the Package system. Suffice to say, within the same Go "Package" you can pretty much drop files in the package directory and Go will just use them. I've written a lot of top-level framework code that accepts user-contributed extensions in languages like Python (see, for example, Graphios), and it's just never as easy as it should be. I'm always having to manually include things at the very least, or more likely, fiddle around with ugly stuff like `__init__py` and `sys.path.insert`.

With PTSD, if you want to implement `VictorOps`, you just make a `victorops.go` file in the package dir and implement the Collector interface. That's pretty much it. The `go build` tooling will pick your file right up along with the rest of it and do the right thing. Literally zero messing around with meta-import magic.

The second reason I like writing modular code is the type system. Types are just so easy to create in Go, and interfaces are, IMO, a much more straightforward way of modeling logic in a systems-programming context than Java-style classes. Go interfaces let you reason about the *functionality* that's common between objects rather than the attributes or data-structures they have in common. In this context, writing a new PTSD Collector means creating a type that implements three simple functions (methods, whatever).

`Enabled` takes no arguments and returns a Boolean indicating whether or not PTSD should enable your collector. This allows us to import all available collector add-ons and just switch on the ones we want. The PagerDuty collector's `Enabled` function returns true if you've set a `PDTOKEN` environment variable containing your PagerDuty token.

`Run` takes an `int` and returns nothing. This `int` is PTSD's polling interval (60 minutes by default). The PagerDuty collector uses this interval to decide how far back it should query the PD API for records of incidents.

Finally, `Name` returns the name of the Collector as a string. This is really just used for debugging (e.g., "enabled collector: <name>").

That's it. Just implement those three functions, and the rest of PTSD knows how to deal with your code, and your code can pretty much do anything it wants vis-à-vis actually collecting metrics and sending them to the Outputters via the global `increment` function. When I'm dealing with class-based systems like Python that use object inheritance, I somewhat ironically wind up creating much more formal and therefore usually less flexible object definitions.

The final reason I chose Go was to make it easy for other people to actually use PTSD in real life. This really boils down to the fact that I get a compiled binary as output so I don't have to worry about moving the toolchain around with the executable. This makes things mind-numbingly simple when I want to embed this in a Docker container or toss it on a Heroku Dyno. Really, I know that this thing I just made is going to just run pretty much wherever I throw it, without any `bundler`, `pip`, `npm`, or any other kind of toolchain dependency hell, and by extension I know that nobody else who wants to use this will have to experience any of that either.

By the time you read this, PTSD should be stable and in use at Librato. Despite the latent scarring, I'm really looking forward to returning to Ops work again, and I hope I've inspired you to think about quantifying the on-call stress in your organization, and possibly giving Go a whirl if you haven't already.

Take it easy.

References:

[1] Monitorama, Joey Parsons talk: https://www.youtube.com/watch?v=1SlljMU9V5k&feature=youtu.be&t=20704).

[2] PTSD (Pager Trauma Statistics Daemon): http://github.com/djosephsen/ptsd.

[3] Anton Holmquist's "Jason" library: https://github.com/antonholmquist/jason.