iVoyeur
Go, in Real Life

DAVE JOSEPHSEN

Dave Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and is Senior Systems Engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his co-authored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project.
dave-usenix@skeptech.org

Through a combination of unfortunate timing, unexpected workload, and laziness, I'm writing this column in the midst of a rare vacation, as I look out on the eastern front of the Rocky Mountains in late fall. I'm using a borrowed laptop (thanks Chris) in a land unencumbered by WiFi, and I'm hoping to find a GPRS signal strong enough to send it to Rik, my editor, before the deadline, which is today I think, or maybe tomorrow.

Although we've arrived only a few weeks later in the year than usual, everything is different here in my favorite place in the world: the air colder, the animals edgier, the light and foliage more dramatic. When we manage to make it up here, we expect to be snowed on at least once or twice, but this time we've been either rained, iced, or snowed on every day. This has only accentuated our hiking, affording us some privacy on the trails, increasing the contrast of our photos, and giving our supposedly waterproof boots an opportunity to prove their worth.

I love the mountains, not just because their size puts humanity in perspective, and not just because they are unabashedly wild. I love the mountains because they encourage good habits in the people who choose to venture into them. They reward hard work, awareness, and respect, and they punish stupidity, sloth, and arrogance. I love the mountains because loving them makes me a better human being.

I had planned this month to write more about libhearsay [1], and show off how I've used it to connect a few different monitoring tools together. But that work is 3000 or so miles away, and anyway those ideas could stand to be baked a bit more before I force them upon you like an excited co-worker with a USB-stick full of vacation slides.

Instead, because I've been writing libhearsay in the Go programming language and also because Go is a newish and hotish programming language created from scratch by the likes of Ken Thompson and Rob Pike, I thought I'd share my experience with it thus far.

In the past few years, many smart programmers have written a bunch of brilliant articles about Go that cover every nook and cranny of every feature and function. None of them, however, seem to convey a sense of what it feels like to create a program in Go, especially from the perspective of a systems guy rather than an application developer. Having worked with it for a few months and a couple of thousand lines, I've noticed that, like the mountains, Go seems to be encouraging beneficial habits in me. Some of these are small things, and are easily articulated, and others are larger and more subtle, but taken together, the patterns, idioms, and manners of thought that Go encourages are making me a better programmer. I think that this, rather than any particular linguistic feature, is the second greatest thing about Go (the greatest thing about it obviously is its enormous potential for name-related puns). Here are a few examples:

Go encourages me to use Git.. ahem, from the git-go

The "go" utility, which is a combination compiler, linker, and packing tool, expects my Go code to be organized into a simple directory structure. If I place a github.com folder inside the top-level src directory of this structure, and commit the contents of a subdirectory of the GitHub folder to GitHub, then other Go users can install and build my program by typing "go get foo" at their command prompt (where "foo" is the name of my project on GitHub).

The go utility will go to GitHub, find my project, clone it into the local users $GOPATH/src/github.com folder, and build it for them. This is pretty great; you get a handy packaging mechanism for free by using revision control, which is something you would have done anyway. It supports sites other than GitHub, such as launchpad, googlecode, and bitbucket, and a slew of version control systems, including Git, Mercurial, Subversion, and Bazaar. You can even use private sites by following a naming convention or by providing a <meta> tag.

The scheme is not without its problems, including, perhaps ironically, that it's not easy to specify upstream package versions, but it's also illustrative of the underlying pragmatism that typifies Go as a language. The developers didn't bother coming up with an unwieldy reimplementation of CPAN or Gems; instead they observed that developers like to keep code in revision control systems and hacked up a simple, lightweight package manager as the shortest path to getting developers what they probably want anyway.

Go encourages me to think about concurrency

Despite the hours (days?) of study I've invested in my considerable understanding of threading models and inter-process/inter-thread communication libraries, and despite the tens (hundreds?) of little test programs I've written in C, Perl, Python, and Ruby in my attempt to implement those models, and even despite the multi-threaded/multi-process open source projects to which I've committed code, I have never once in my professional life written a concurrent program for use in production. Nor have I ever revisited and rewritten one of the thousands of little tools I've written to make it concurrent. Not that is, until I met Go.

This is not for lack of understanding or caring on my part. In real life I'm an OPS, and the nature of the job just makes impractical the creation of multi-threaded tools to solve the mundane sort of everyday problems that I run into (at least in the shops I've worked in so far). There is neither the time nor the payoff. This sucks for me, because it means I don't get to think concurrently often, and as I grow older, it probably renders that sort of thinking more difficult for me. So that's awesome; my current languages are destroying my brain.

The second Go program I ever wrote was concurrent. It was not concurrent because I wanted to prove or understand the model, or because I was bound and determined to use go-routines and channels. It became concurrent naturally, as a result of my problem and the fact that go-routines were available. Go-routines are so handy that functionally, their use is hard to avoid. Which brings me to:

Go encourages me to network

In the past, for example, I would avoid putting socket code into my tools. I've written socket programs for my own edification, and fully understand the threading issues among others, but in real life it almost always makes more sense to quickly hack up something to standard I/O and rely on daemontools, for example, for TCP. This sentiment is alive and well among the node.js crowd these days, but it is simply no longer true with Go. The concurrency features are so well implemented that there is no reason not to roll your own TCP server.

For anything of moderate size that is expected to remain resident in memory, there's no reason not to roll your own HTTP server for that matter, and it's pretty common practice among Go developers to build something like a distributed worker daemon in Go, and then add an HTTP server to it to export metrics and state data, or add an interface to control the worker remotely.

Go encourages me to embrace type and think about data structures

In Go, creating your own type and extending it with a method is so simple that even as someone who has never been enamored of OOP, or the concept of sub-classing, I find myself naturally reasoning about my solutions primarily in terms of the interaction between custom types. I think Go makes this palatable to me because there isn't any ceremony or magic involved. Type creation is no different from typedeffing in C, and adding methods to types is only trivially different from function declaration.

As a result, where in any other language I might create an array of doohickeys, and loop across them doing whatever, like:

```
for(i=0, i<numberOfDoohickeys,i++) myDoohickey=listOfThings[i]
doWhatever(myDoohickey)
```

in Go I'm much more likely to create a doohickey type of my own to store in the array (which is probably a pretty complex (for me) nested type), which has a built-in whatever method like this:

```
for i in listOfDoohickeys i.Whatever
```

I know, those pretty much seem like the same thing, but by creating my own doohickey I get to think about lots of interesting things, such as exactly how large a doohickey is in memory and whether the system creates a copy of my doohickey in memory when it performs the whatever function, or operates directly on the existing doohickey via a pointer.

It also means that, although a program that loops across some doohickeys doing whatever is useful maybe once or twice, a program that defines doohockeys and implements an interface to them that does whatever is useful may be a lot longer, because other developers (or I) can come back later and trivially add more interfaces to do other things. Now we have a shop-wide means of dealing with doohickeys, and everybody who does whatever to a doohickey from now on will do it in a repeatable way without having to reinvent the wheel.

There's an xkcd comic [2] where, having been asked to pass the salt, an off-frame OCDish person begins developing a general interface that will enable him to pass arbitrary condiments, and over-engineering like this can easily get out of hand in some of the other languages I've used. But I've noticed that general interfaces spring into being quite naturally in Go without any grand intention or purpose on my part; I didn't whiteboard an interface for doohickeys, or prototype it in a simple language and then properly reimplement it in another. I didn't begin by creating a doohickey library or subclassing something doohickey-like. I—a meathead, knuckle-dragging OPS—in scratching my own immediate doohickey itches, tend to accidently create robust, probably even concurrent engineering solutions in Go. Solutions that other OPS are likely to thank me for. As someone who has, for years, prefaced my scripts with something like:

```
#Blame Dave: Fri Sep 15 20:56:47 CDT 2006
```

I appreciate creating code that I don't need to feel vaguely guilty about.

Finally, in other languages I've used, a certain amount of risk came along with simplifying things like sockets; a linear relationship between the language's ability to expose cool features and the amount of cruft in my own code as I bolted on this or that. I had to keep things simple, so the program execution remained knowable—and this is perhaps unfortunate, because what is the point of having a simple interface to sockets if you always feel like it's too cognitively expensive or ugly and bloated to use?

In Go, however, the type system has a tendency to keep everything clean and compartmentalized. My Go code is resistant to cruft. If you aren't fighting it, the code naturally segments and documents itself via its type and function definitions, so adding something like a TCP server doesn't clutter things up, and more importantly, doesn't make your types—and therefore your program—any more difficult to reason about. To be clear, I'm not throwing HTTP servers into everything I write just in case, but I'm certainly more likely to add something like a network interface to expose some analytics where it makes sense to do so.

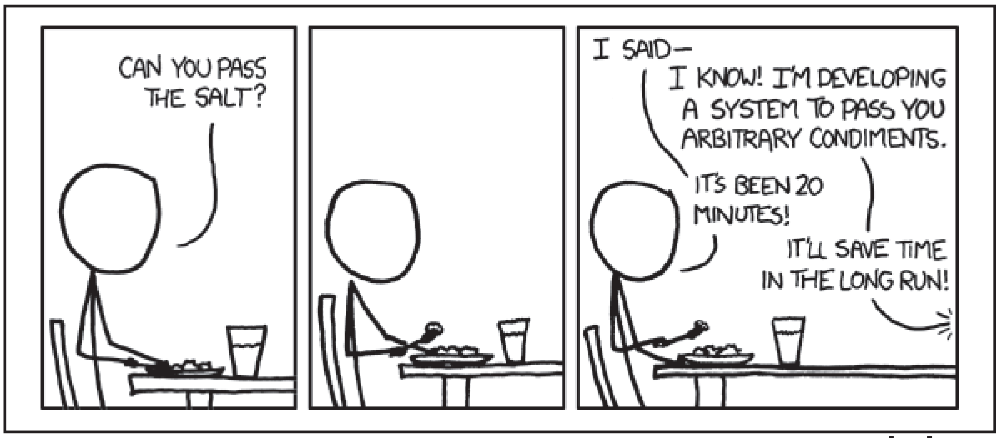
I'm painfully aware that most of what I've said in this article amounts to subjective drivel that could probably be repeated en masse by any proponent of any programming language ever, so even though it won't help, I'll mention that I'm not married to Go and, in fact, program in a multitude of languages. My intent here was not to steal anyone's mindshare or compliment Go at the expense of any other language in particular. But I will wholeheartedly suggest that you learn Go if you get a chance. If you start using it, I think you'll notice that Go wants you to be productive. It keeps things simple, stays out of your way, rewards you for being you, empowers you to build interesting stuff, and makes you a better programmer in the process.

References

[1] libhearsay http://www.skeptech.org/hearsay.

[2] http://xkcd.com/974/.

xkcd


CAN YOU PASS THE SALT?
I SAID—
I KNOW! I'M DEVELOPING A SYSTEM TO PASS YOU ARBITRARY CONDIMENTS.
IT'S BEEN 20 MINUTES!
IT'LL SAVE TIME IN THE LONG RUN!

xkcd.com

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Buy the Box Set!

Whether you had to miss a conference, or just didn't make it to all of the sessions, here's your chance to watch (and re-watch) the videos from your favorite USENIX events. Purchase the "Box Set," a USB drive containing the high-resolution videos from the te chnical sessions. This is perfect for folks on the go or those without consistent Internet access.

Box Sets are available for:

- **LISA '13:** 27th Large Installation System Administration Conference
- **USENIX Security '13:** 22nd USENIX Security Symposium
- **HealthTech '13:** 2013 USENIX Workshop on Health Information Technologies
- **WOOT '13:** 7th USENIX Workshop on Offensive Technologies
- **UCMS '13:** 2013 USENIX Configuration Mangement Summit
- **HotStorage '13:** 5th USENIX Workshop on Hot Topics in Storage and File Systems
- **HotCloud '13:** 5th USENIX Workshop on Hot Topics in Cloud Computing
- **WiAC '13:** 2013 USENIX Women in Advanced Computing Summit
- **NSDI '13:** 10th USENIX Symposium on Networked Systems Design and Implementation
- **FAST '13:** 11th USENIX Conference on File and Storage Technologies
- **LISA '12:** 26th Large Installation System Administration Conference

Learn more at:
www.usenix.org/boxsets