Protobuffers Are Wrong

2018-10-10

I've spent a good deal of my professional life arguing against using protobuffers. They're clearly written by amateurs, unbelievably ad-hoc, mired in gotchas, tricky to compile, and solve a problem that nobody but Google really has. If these problems of protobuffers remained quarantined in serialization abstractions, my complaints would end there. But unfortunately, the bad design of protobuffers is so persuasive that these problems manage to leak their way into your code as well.

Ad-Hoc and Built By Amateurs

Stop. Put away your email client that is half-way through writing me about how "Google is filled with the world's best engineers," and that "anything they build is, by definition, not built by amateurs." I don't want to hear it.

Let's just get this out of the way. Full disclosure: I used to work at Google. It was the first (but unfortunately, not the last) place I ever used protobuffers. All of the problems I want to talk about today exist inside of Google's codebase; it's not just a matter of "using protobuffers wrong" or some such nonsense like that.

By far, the biggest problem with protobuffers is their terrible type-system. Fans of Java should feel right at home with protobuffers, but unfortunately, literally nobody considers Java to have a well-designed type-system. The dynamic typing guys complain about it being too stifling, while the static typing guys like me complain about it being too stifling without giving you any of the things you actually want in a type-system. Lose lose.

The ad-hoc-ness and the built-by-amateurs-itude go hand-in-hand. So much of the protobuffer spec feels bolted on as an afterthought that it clearly *was* bolted on as an afterthought. Many of its restrictions will make you stop, scratch your head and ask "wat?" But these are just symptoms of the deeper answer, which is this:

Protobuffers were obviously built by amateurs because they offer *bad solutions to widely-known and already-solved problems.*

No Compositionality

Protobuffers offer several "features", but none of them see to work with one another. For example, look at the list of orthogonal-yet-constrained typing features that I found by skimming the documentation.

- `oneof` fields can't be `repeated`.
- map<k,v> fields have dedicated syntax for their keys and values, but this isn't used for any other types.
- Despite `map` fields being able to be parameterized, no user-defined types can be. This means you'll be stuck hand-rolling your own specializations of common data structures.
- `map` fields cannot be `repeated`.
- `map` keys *can* be `string`s, but *can not* be `bytes`. They also can't be `enum`s, even though `enum`s are considered to be equivalent to integers everywhere else in the protobuffer spec.
- `map` values cannot be other `map`s.

This insane list of restrictions is the result of unprincipled design choices and bolting on features after the fact. For example, `oneof` fields can't be `repeated` because rather than resulting in a coproduct type, instead the code generator will give you a product of mutually-exclusive optional fields. Such a transformation is only valid for a singular field (and, as we'll see later, not even then.)

The restriction behind `map` fields being unable to be `repeated` is related, but shows off a different limitation of the type-system. Behind the scenes, a map<k,v> is desugared into something spiritually similar to repeated Pair<k,v>. And because `repeated` is a magical language keyword rather than a type in its own right, it doesn't compose with itself.

Your guess is as good as mine for why an `enum` can't be used as a `map` key.

What's so frustrating about all of this is a little understanding of how modern type-systems work would be enough to *drastically simplify* the protobuffer spec and simultaneously *remove all of the arbitrary restrictions.*

The solution is as follows:

- Make all fields in a message `required`. This makes messages *product types*.
- Promote `oneof` fields to instead be standalone data types. These are *coproduct types*.
- Give the ability to parameterize product and coproduct types by other types.

That's it! These three features are all you need in order to define any possible piece of data. With these simpler pieces, we can re-implement the rest of the protobuffer spec in terms of them.

For example, we can rebuild `optional` fields:

```
product Unit {
  // no fields
}

coproduct Optional<t> {
  t    value = 0;
  Unit unset = 1;
}
```

Building `repeated` fields is simple too:

```
coproduct List<t> {
  Unit empty = 0;
  Pair<t, List<t>> cons = 1;
}
```

Of course, the actual serialization logic is allowed to do something smarter than pushing linked-lists across the network—after all, implementations and semantics don't need to align one-to-one.

Questionable Choices

In the vein of Java, protobuffers make the distinction between *scalar* types and *message* types. Scalars correspond more-or-less to machine primitives—things like `int32`, `bool` and `string`. Messages, on the other hand, are everything else. All library- and user-defined types are messages.

The two varieties of types have completely different semantics, of course.

Fields with scalar types are always present. Even if you don't set them. Did I mention that (at least in proto3[1]) all protobuffers can be zero-initialized with absolutely no data in them? Scalar fields get false-y values—`uint32` is initialized to `0` for example, and `string` is initialized as `""`.

It's impossible to differentiate a field that was missing in a protobuffer from one that was assigned to the default value. Presumably this decision is in place in order to allow for an optimization of not needing to send default scalar values over the wire. Presumably, though the encoding guide makes no mention of this optimization being performed, so your guess is as good as mine.

As we'll see when we discuss protobuffers' claim to being god's gift to backwards- and forwards-compatible APIs, this inability to distinguish between unset and default values is a nightmare. Especially if indeed it's a design decision made in order to save one bit (set or not) per field.

Contrast this behavior against message types. While scalar fields are dumb, the behavior for message fields is outright *insane.* Internally, message fields are either there or they're not—but their behavior is crazy. Some pseudocode for their accessor is worth a thousand words. Pretend this is Java or something similar:

```
private Foo m_foo;

public Foo foo {
  // only if `foo` is used as an expression
  get {
    if (m_foo != null)
      return m_foo;
    else
      return new Foo();
  }

  // instead if `foo` is used as an lvalue
  mutable get {
    if (m_foo = null)
      m_foo = new Foo();
    return m_foo;
  }
}
```

The idea is that if the `foo` field is unset, you'll see a default-initialized copy whenever you ask for it, but won't actually modify its container. But if you modify `foo`, it will modify its parent as well! All of this just to avoid using a `Maybe Foo` type and the associated "headaches" of the nuance behind needing to figure out what an unset value should mean.

This behavior is especially egregious, because it breaks a law! We'd expect the assignment `msg.foo = msg.foo;` to be a no-op. Instead the implementation will actually silently change `msg` to have a zero-initialized copy of `foo` if it previously didn't have one.

Unlike scalar fields, at least it's possible to detect if a message field is unset. Language bindings for protobuffers offer something along the lines of a generated `bool has_foo()` method. In the frequent case of copying a message field from one proto to another, iff it was present, you'll need to write the following code:

```
if (src.has_foo(src)) {
  dst.set_foo(src.foo());
}
```

Notice that, at least in statically-typed languages, this pattern *cannot be abstracted* due to the nominal relationship between the methods `foo()`, `set_foo()` and `has_foo()`. Because all of these functions are their own *identifiers*, we have no means of programmatically generating them, save for a preprocessor macro:

```
#define COPY_IFF_SET(src, dst, field) \
if (src.has_##field(src)) { \
  dst.set_##field(src.field()); \
}
```

(but preprocessor macros are verboten by the Google style guide.)

If instead all optional fields were implemented as `Maybe`s, you'd get abstract-able, referentially transparent call-sites for free.

To change tack, let's talk about another questionable decision. While you can define `oneof` fields in protobuffers, their semantics are *not* of coproduct types! Rookie mistake my dudes! What you get instead is an optional field for each case of the `oneof`, and magic code in the setters that will just unset any other case if this one is set.

At first glance, this seems like it should be semantically equivalent to having a proper union type. But instead it is an accursed, unutterable source of bugs! When this behavior teams up with the law-breaking implementation of `msg.foo = msg.foo;`, it allows this benign-looking assignment to silently delete arbitrary amounts of data!

What this means at the end of the day is that `oneof` fields do not form law-abiding `Prism`s, nor do messages form law-abiding `Lens`es. Which is to say good luck trying to write bug-free, non-trivial manipulations of protobuffers. It is *literally impossible to write generic, bug-free, polymorphic code over protobuffers.*

That's not the sort of thing anybody likes to hear, let alone those of us who have grown to love parametric polymorphism—which gives us the *exact opposite promise.*

The Lie of Backwards- and Forwards-Compatibility

One of the frequently cited killer features of protobuffers is their "hassle-free ability to write backwards- and forwards-compatible APIs." This is the claim that has been pulled over your eyes to blind you from the truth.

What protobuffers are is *permissive.* They manage to not shit the bed when receiving messages from the past or from the future because they make absolutely no promises about what your data will look like. Everything is optional! But if you need it anyway, protobuffers will happily cook up and serve you something that typechecks, regardless of whether or not it's meaningful.

This means that protobuffers achieve their promised time-traveling compatibility guarantees by *silently doing the wrong thing by default.* Of course, the cautious programmer can (and should) write code that performs sanity checks on received protobuffers. But if at every use-site you need to write defensive checks ensuring your data is sane, maybe that just means your deserialization step was too permissive. All you've managed to do is decentralize sanity-checking logic from a well-defined boundary and push the responsibility of doing it throughout your entire codebase.

One possible argument here is that protobuffers will hold onto any information present in a message that they don't understand. In principle this means that it's nondestructive to route a message through an intermediary that doesn't understand this version of its schema. Surely that's a win, isn't it?

Granted, on paper it's a cool feature. But I've never once seen an application that will actually preserve that property. With the one exception of routing software, nothing wants to inspect only some bits of a message and then forward it on unchanged. The vast majority of programs that operate on protobuffers will decode one, transform it into another, and send it somewhere else. Alas, these transformations are bespoke and coded by hand. And hand-coded transformations from one protobuffer to another don't preserve unknown fields between the two, because it's literally meaningless.

This pervasive attitude towards protobuffers always being compatible rears its head in other ugly ways. Style guides for protobuffers actively advocate against DRY and suggest inlining definitions whenever possible. The reasoning behind this is that it allows you to evolve messages separately if these definitions diverge in the future. To emphasize that point, the suggestion is to fly in the face of 60 years' worth of good programming practice just in case *maybe* one day in the future you need to change something.

At the root of the problem is that Google conflates the meaning of data with its physical representation. When you're at Google scale, this sort of thing probably makes sense. After all, they have an internal tool that allows you to compare the finances behind programmer hours vs network utilization vs the cost to store x bytes vs all sorts of other things. Unlike most companies in the tech space, paying engineers is one of Google's smallest expenses. Financially it makes sense for them to waste programmers' time in order to shave off a few bytes.

Outside of the top five tech companies, none of us is within five orders of magnitude of being Google scale. Your startup *cannot afford* to waste engineer hours on shaving off bytes. But shaving off bytes and wasting programmers' time in the process is exactly what protobuffers are optimized for.

Let's face it. You are not Google scale and you never will be. Stop cargo-culting technology just because "Google uses it" and therefore "it's an industry best-practice."

Protobuffers Contaminate Codebases

If it were possible to restrict protobuffer usage to network-boundaries I wouldn't be nearly as hard on it as a technology. Unfortunately, while there are a few solutions in principle, none of them is good enough to actually be used in real software.

Protobuffers correspond to the data you want to send over the wire, which is often *related* but not *identical* to the actual data the application would like to work with. This puts us in the uncomfortable position of needing to choose between one of three bad alternatives:

1. Maintain a separate type that describes the data you actually want, and ensure that the two evolve simultaneously.
2. Pack rich data into the wire format for application use.
3. Derive rich information every time you need it from a terse wire format.

Option 1 is clearly the "right" solution, but its untenable with protobuffers. The language isn't powerful enough to encode types that can perform double-duty as both wire and application formats. Which means you'd need to write a completely separate datatype, evolve it synchronously with the protobuffer, and *explicitly write serialization code between the two.* Seeing as most people seem to use protobuffers in order to not write serialization code, this is obviously never going to happen.

Instead, code that uses protobuffers allows them to proliferate throughout the codebase. True story, my main project at Google was a compiler that took "programs" written in one variety of protobuffer, and spit out an equivalent "program" in another. Both the input and output formats were expressive enough that maintaining proper parallel C++ versions of them could never possibly work. As a result, my code was unable to take advantage of any of the rich techniques we've discovered for writing compilers, because protobuffer data (and resulting code-gen) is simply too rigid to do anything interesting.

The result is that a thing that could have been 50 lines of recursion schemes was instead 10,000 lines of ad-hoc buffer-shuffling. The code I wanted to write was literally impossible when constrained by having protobuffers in the mix.

While this is an anecdote, it's not in isolation. By virtue of their rigid code-generation, manifestations of protobuffers in languages are never idiomatic, nor can they be made to be—short of rewriting the code-generator.

But even then, you still have the problem of needing to embed a shitty type-system into the targeted language. Because most of protobuffers' features are ill-conceived, these unsavory properties leak into our codebases. It means we're forced to not only implement, but also use these bad ideas in any project which hopes to interface with protobuffers.

While it's easy to implement inane things out of a solid foundation, going the other direction is challenging at best and the dark path of Eldrich madness at worst.

In short, abandon all hope ye who introduce protobuffers into your projects.

---

1. To this day, there's a raging debate inside Google itself about proto2 and whether fields should ever be marked as `required`. Manifestos with both titles "`optional` considered harmful" *and* "`required` considered harmful." Good luck sorting that out. ↩

REASONABLY POLYMORPHIC

Hi there. I'm **Sandy Maguire**. I like improving life and making cool things.

If you want to get in touch, I'd love to hear from you! Send me an email; you can contact me via **sandy** at **sandymaguire.me**.