« The Road to OCIv2 Images: What's Wrong with Tar?

Aleksa Sarai

containers oci ociv2-images rant 21 January 2019

You might not realise it yet, but you very likely want a better container image format than the ones currently available. How can I say this with such confidence? Because the current design of container images provides almost none of the properties that most people likely want from container images – even the properties you *think* you have aren't omnipresent. The root cause of this problem is an unlikely and seemingly innocent suspect – `tar`. But don't let its appearance and familiarity fool you, `tar` isn't the format you might think it is.

This is the first of a series of articles I will be writing over the coming weeks, which will outline an improvement to the Open Container Initiative (OCI) image format in order to provide the properties that you want.

My first instinct was to title this article "Tar Considered Harmful", but I had a feeling that the peanut gallery would cringe at such a title. However, this article is very much a foundational discussion of `tar` and how it fundamentally fails in the use-case of container images (which will outline what we really want from a container image format). There have been some other articles that touch on the issues I go over here, but I hope I can provide a more cohesive insight into the issues with `tar`. Then again, some folks believe that `tar` is the best format for container images. I hope this first article will serve as a decent rebuttal.

I am currently working on a proof-of-concept of these ideas, and hopefully it will be ready soon for inclusions into umoci (a generic OCI image manipulation tool, which is being used by quite a few folks now-a-days for building and operating on container images).

But first, I guess we should make sure we're on the same page on what container images actually are.

What [Container Images] Are Made Of

Copyright © Aleksa Sarai 2014-2019. This website is free software and is licensed under the GNU AGPLv3+. Source code.

> *host kernel, but in a context where the filesystem (among other system resources) are virtualised. (To paraphrase Bryan Cantrill, "virtual" in this context is just a more diplomatic way of saying "lie".)*

Because the filesystem is virtualised, we need to have a different root filesystem than our host (just like a `chroot`, which one could argue was the first instance of "something like a container"). Obviously a root filesystem is just a directory (or some union filesystem that looks like a directory at the end of the day). But then the follow-on question is "how are we going to distribute these directories?" – and that's what a container image is.

Currently there are basically two models of container images:

- Layered container images. These are what most people think of when you say "container image", and are made up of a series of layers which are stacked on top of each other during container execution (with a union filesystem or similar tricks). These layers are almost always `tar` archives, and is the beginning of the problem. Usually such images also contain several bits of metadata (usually JSON).
- "Flat" container images. You can think of these as being more analogous to VMs. The most obvious examples of such container images are LXD's images (which are effectively a single archive) and Singularity (which passes around a full filesystem image, which gets loopback mounted).

Most of the issues I will go over only really apply to layered container images. Due to the their design, "flat" container images have fewer problems (mainly because they aren't interested in some of the features you get from layered images). In the case of LXD, their design actually handles some of these concerns anyway (in particular, "transfer de-duplication" isn't necessary because they use binary deltas for updating images – and images are auto-updated by default on LXD).

This article will be focusing on OCI (Open Container Initiative) images, because that's the standardised container image format (and I really hope it will get wider use if we can provide a clear advantage over other image formats). However, the same issues apply verbatim to Docker images – the OCI image format was based directly on the on-disk Docker format.

Just to make sure you get what an OCI image looks like, here is what it looks like after you've downloaded one (skopeo is a tool which translates images between formats, and also supports fetching images):

```
% skopeo copy docker://rust:latest oci:rust:latest
Getting image source signatures
Copying blob sha256:54f7e8ac135a5f502a6ee9537ef3d64b1cd2fa570dc0a40b4d3b6f7ac81e7486
 43.22 MB / 43.22 MB [======================================================] 4s
Copying blob sha256:d6341e30912f12f56e18564a3b582853f65376766f5f9d641a68a724ed6db88f
 10.24 MB / 10.24 MB [======================================================] 1s
Copying blob sha256:087a57faf9491b1b82a83e26bc8cc90c90c30e4a4d858b57ddd5b4c2c90095f6
```

```
 4.14 MB / 4.14 MB [=========================================================] 0s
Copying blob sha256:5d71636fb824265e30ff34bf20737c9cdc4f5af28b6bce86f08215c55b89bfab
 47.74 MB / 47.74 MB [=======================================================] 4s
Copying blob sha256:0c1db95989906f161007d8ef2a6ef6e0ec64bc15bf2c993fd002edbdfc7aa7df
 203.34 MB / 203.34 MB [=====================================================] 20s
Copying blob sha256:734ee16af2dd89c09a46ff408ffc44679aca2e1b8a10baec4febd9a7b6ac9778
 218.11 MB / 218.11 MB [=====================================================] 41s
Copying config sha256:af2dafa4b223aa1ab6ca6f6c35c5fce093254602cff4b2a8429850764d533b29
 4.14 KB / 4.14 KB [=========================================================] 0s
Writing manifest to image destination
Storing signatures

% tree rust/
rust/
├── blobs
│   └── sha256
│       ├── 087a57faf9491b1b82a83e26bc8cc90c90c30e4a4d858b57ddd5b4c2c90095f6
│       ├── 0c1db95989906f161007d8ef2a6ef6e0ec64bc15bf2c993fd002edbdfc7aa7df
│       ├── 2696f7292a958d02760e3a8964e554a3a6176fb7e04fc66be8760b3b05cbe65b
│       ├── 54f7e8ac135a5f502a6ee9537ef3d64b1cd2fa570dc0a40b4d3b6f7ac81e7486
│       ├── 5d71636fb824265e30ff34bf20737c9cdc4f5af28b6bce86f08215c55b89bfab
│       ├── 734ee16af2dd89c09a46ff408ffc44679aca2e1b8a10baec4febd9a7b6ac9778
│       ├── af2dafa4b223aa1ab6ca6f6c35c5fce093254602cff4b2a8429850764d533b29
│       └── d6341e30912f12f56e18564a3b582853f65376766f5f9d641a68a724ed6db88f
├── index.json
└── oci-layout

2 directories, 10 files

% find rust/ -type f | xargs file -z
rust/blobs/sha256/54f7e8ac135a5f502a6ee9537ef3d64b1cd2fa570dc0a40b4d3b6f7ac81e7486: POSI⟩
rust/blobs/sha256/d6341e30912f12f56e18564a3b582853f65376766f5f9d641a68a724ed6db88f: POSI⟩
rust/blobs/sha256/087a57faf9491b1b82a83e26bc8cc90c90c30e4a4d858b57ddd5b4c2c90095f6: POSI⟩
rust/blobs/sha256/5d71636fb824265e30ff34bf20737c9cdc4f5af28b6bce86f08215c55b89bfab: POSI⟩
rust/blobs/sha256/0c1db95989906f161007d8ef2a6ef6e0ec64bc15bf2c993fd002edbdfc7aa7df: POSI⟩
rust/blobs/sha256/734ee16af2dd89c09a46ff408ffc44679aca2e1b8a10baec4febd9a7b6ac9778: POSI⟩
rust/blobs/sha256/af2dafa4b223aa1ab6ca6f6c35c5fce093254602cff4b2a8429850764d533b29: JSON
rust/blobs/sha256/2696f7292a958d02760e3a8964e554a3a6176fb7e04fc66be8760b3b05cbe65b: JSON
rust/oci-layout:                                                                  JSON
rust/index.json:                                                                  JSON
```

J’accuse! There are our `tar` archives, and you’ll notice that there’s one for each layer. There are also some JSON blobs, which aren’t really of interest to us here. There is also another important point to notice – OCI images use a content-addressable store as their backbone storage mechanism (`index.json` is an “entry point” to the store – and is what contains the tags in most cases).

It should also be noted that OCI images all use “smart pointers” (that is, “pointers”

which contain the content-addressable digest of the target as well as its **media-type** and size) which you can see in `index.json`:

```
% jq '.manifests[0]' rust/index.json
{
  "mediaType": "application/vnd.oci.image.manifest.v1+json",
  "digest": "sha256:2696f7292a958d02760e3a8964e554a3a6176fb7e04fc66be8760b3b05cbe65b",
  "size": 1146,
  "annotations": {
    "org.opencontainers.image.ref.name": "latest"
  },
  "platform": {
    "architecture": "amd64",
    "os": "linux"
  }
}
```

These facts will become important later, when we talk about a new container image format that is built on top of the OCI content-addressable store (smart pointers and the ability to add new media-types will help us out).

As an aside, if you've ever wanted to know what the best container image format is, the short answer is basically "none of them". The problem is that almost all of them have nice features that the others could really use, but because everyone wants to work on their own thing there's much less cross-pollination than you'd like. Examples of such features include LXC's templates, OCI's content-addressability, or AppC's dependencies.

What Has [Tar] Done For Us Lately?

`tar` is a very old format, having been born within the original Unix source, and thus there is a lot of history within it. And, to no-one's surprise, it's a pretty ugly format in many ways. It simply doesn't match what we need in a container image format, and I would argue it barely matches what most people today need in an archive format (though that's out-of-scope for now).

Now, don't misunderstand what I'm saying – my point here is not "it's old, so it's bad." `tar` is the obvious choice for an archive format, due to its long history and ubiquity, and writing a custom format with no justification would be a borderline reckless thing to do. However, `tar`'s history is important to understanding how it got to the state it's in today. This section will be quite long-winded (there's forty-something years of history to distil into a single blog post), but you can skip to the end.

This is not necessarily a new idea or argument, other folks have voiced similar concerns. I'm hoping that I can provide a cohesive overview of both `tar`'s generic issues as well as how its usage is even worse in the context of container images.

The full history lesson is a bit long (and probably something I'm not qualified to give) so I'll just give you the highlights – if you'd like more in-depth information you can always take a look at `pax(1)`, `tar(5)`, `star(5)` and the GNU tar internals documentation. This OCI PR discussion is a good example of how much back-and-forth can come about when discussing what an image specification "really means" when it says "`tar`".

Genesis

`tar` first originated in Unix v7. Curiously, it was not the first archiving tool available for Unix. `tar` was a successor to `tp` (Unix v4), which itself was a successor to `tap` (Unix v1). As a complete aside, this appears to be the reason why `tar` accepts dash-less arguments (such as `tar xvf`). Unix v1 didn't have dashed argument flags like `-xvf` (as far as I can tell from the man pages), and `tar` appears to have been backwards-compatible with `tp` (which was backwards-compatible with `tap`). Therefore the most likely reason why `tar` supports dash-less arguments is because some folks in the 70s wanted to be able to `alias tap=tp tp=tar` and it's stuck ever since. This should tell you what the primary theme of these history sections will be.

But that's all besides the point. `tar` was introduced in Unix v7, as a format for storing data on *tape archives*. It didn't support compression. It didn't even contain a magic header (so `file foo.tar` couldn't always tell you the file type). The design of the format was very simple, with fixed-length (512-byte) headers that contained all the information you might expect (file name, a single-byte "entry type", length, mode, owner, modified time, and so on). If there was a non-zero length then it was followed by the contents of the file. This basic structure of `tar` archives has been retained over the past 40 years.

To say that the format was strangely designed would be an understatement. First of all, all of the numerical values were stored in **octal ASCII** – which artificially limited the maximum entry size to about 8GB. In addition, symlinks (and hardlinks) were handled by storing the "link name" in the fixed-length header – resulting in each header containing 100 NUL bytes unless it was a symlink or hardlink. Obviously the pathname was restricted, but the restriction was exceptionally peculiar – rather than restricting the total pathname to 255 bytes, they restricted the `basename` of the path to 100 bytes and the `dirname` to 155 bytes (meaning that long `dirnames` and long `basenames` were both forbidden needlessly – and strangely `tar` calls `dirname` "prefix"). Curiously, the final 12 bytes of this 512-byte header remain unused in any standard to this day (it has been used by Solaris's `tar` as well as `star`, but these are extensions).

Very soon, people started extending the original tar. The history of this is quite

complicated, and `tar` definitely went through all of the Unix wars (in a way, it's a looking-glass for the history of Unix). Long before POSIX.1-1988 (which introduced `ustar`) came around, there were a few competing implementations. Solaris's `tar`, FreeBSD's `bsdtar`, GNU's `tar`, and Jörg Schilling's `star` are the most notable. There was some cross-over between these different implementations, but eventually you ended up with a hodgepodge of different `tar`-like archive formats (usually the same feature was re-implemented in different ways by different implementations). And, at the request of users, most of `tar` implementations were forced to become somewhat interoperable with all of these other formats.

It's important to keep in mind where `tar` comes from to understand why its use is no longer reasonable.

The [Extension] Wars

Before we get into the flurry of extensions (and POSIX's inability to contain them), I should probably explain how you might extend a `tar` archive. I mentioned above that each `tar` header contains a one-byte "entry type". This is the primary way that extensions operate. All of the built-in entry types were ASCII decimals (with the exception of an ordinary file which could be represented either as a NUL byte or as `'0'`). Before POSIX.1-1988, this was essentially a free-for-all, with various vendors coming up with their own custom header extensions as well as creating their own wacky entry types. It was definitely a fun time.

The most obvious things to extend should be pretty apparent – the limited size as well as pathname restrictions. GNU `tar` partially fixed the size problem by storing size in "base-256", and created special "long name" entry types that allowed you to have files (and links) with arbitrarily long pathnames. Sparse file support was added too, with varying degrees of support by other implementations (recall that interoperability requires everyone else to implement your special feature too).

Then the push for a "standard Unix for the masses" came along in the form of POSIX and the eventual release of the first edition of the standard, POSIX.1-1988. And `tar` was included as part of this specification, with a new format called `ustar` (**U**nix **S**tandard **TAR**) that was meant to be the one format to rule them all. Unfortunately (like most things in POSIX) there was a need to placate every Unix vendor, and so the specification was incredibly generic and basic in terms of the features it defined. Most of the core properties of `tar` were unchanged, though some quite important changes were made. For instance, POSIX.1-1988 requires all `ustar` archives to set `ustar\0` as the `magic` field (and to add an empty header with just `magic` set to `ustar\0` at the beginning of the archive) so that tools like `file` can actually reliably recognise `ustar` archives (as I mentioned above, before this change, *there was no*

reliable way of detecting whether something was a `tar` archive).

Unfortunately, one of the largest problems with `tar` compatibility was left woefully underspecified in POSIX.1-1988's `ustar` – how extensions should be handled between vendors. This was an issue that had caused lots of compatibility troubles in the past because implementations couldn't recognise that the strange header they're parsing was actually a foreign extension they didn't support). The only "extension" handling that was provided by POSIX.1-1988 was that vendors could use any upper-case letter (all 26 of them) to store their own implementation-defined extensions and headers. After all, who would need more than 26 extensions – right?

Unsurprisingly, this didn't help resolve the issue at all. GNU, Solaris, `star`, and several others started using up this very limited namespace for a variety of their own extensions. As I mentioned above, file names and link targets were hideously restricted in length, and so GNU used `L` and `K` (not to mention their previous usage of `N` in the old GNU format) for this purpose. And so the namespace became saturated with all of these different extensions, with people being worried about conflicts between different implementations – a rather odd example is that the "POSIX.1-2001 eXtended" format uses `X` as an extension header, despite this header having been used by Solaris for a very long time. Another quite problematic conflict is that both GNU `tar` and `star` used `S` to represent sparse files, but had slightly different semantics which usually ended *brilliantly*. So everyone went back to supporting everyone else's extensions to keep users happy, and POSIX pretty much sat on their hands in this department until 2001.

Interestingly, as far as I can tell, we never actually used up all 26 extensions slots. But there were still a bunch of conflicts within the slots that were used (such as `S` sparse file support). I guess you could argue this is a side-effect of the Birthday "Paradox" or we're just really bad at sharing resources between different implementations.

PAX: A New [Standard]

In 2001, POSIX declared that enough was enough. It was clear that `ustar` hadn't solved the issues that they'd hoped to solve (vendor compatibility and modernising the Unix v7 `tar` format). In addition, the `tar` vs. `cpio` war hadn't fizzled out – and POSIX wanted to have *One Format To Rule Them All™*. So POSIX.1-2001 scrapped `cpio` and `ustar` and came up with a new format, called PAX (apparently "pax" is meant to to be a pun, since it means "peace" in Latin – and the intention of PAX was to bring peace between the `tar` and `cpio` camps).

PAX is effectively `ustar` but with a series of extensions that they hoped would alleviate some of the issues that weren't fixed by `ustar`. While POSIX might refer to

PAX as being a different format from `tar`, when someone these days uses the word "tar" they usually are referring to PAX. The only thing PAX stole from `cpio` is its *lovely* command-line argument design in the POSIX-defined tool `pax` (which was meant to replace the need for `tar` and `cpio` – though of course `tar` just ended up supporting PAX, `cpio` is still alive and kicking, and almost nobody has even heard of `pax`).

The primary extension was the addition of "pax Header Blocks", which is a pair of new entry types that allow for key-value metadata to be applied for a given `ustar` entry (`x` applies it to the next entry, while `g` applies the metadata to the entire archive and must appear at the start of the archive). The metadata is stored as the "file contents" of the entry, with each key-value mapping being stored as `key=value` (separated by NUL bytes). A variety of keywords were defined as part of PAX, which deprecated older vendor extensions (examples include long names with `path` and `linkpath`, large file sizes with `size`, as well as support for custom text encodings with `hdrcharset` and `charset`). Interestingly, pre-PAX there was no standard way to represent the `atime` or `ctime` of a file since the Unix v7 header only had a field for `mtime`. PAX "resolved" this issue for the most part, though see the next section for more details.

Another interesting extension was to add an end-of-archive delimiter, which is two empty 512-byte headers (meaning all PAX-compliant `tar` archives have a 1K blank footer).

For extensions they decided to create a much more fully-fledged extension system than existed in `ustar`. Keeping with the theme of "uppercase ASCII is vendor space", they allowed vendors to use keywords in the form `<VENDOR>.<keyword>` (with `<VENDOR>` being a vendor-specific name in all-caps). This opened the door to arbitrarily many vendor-specific extensions – with each vendor using their own namespace! This is nice, though as we'll see in a minute, it did come with some downsides.

All-in-all, PAX was a fairly large improvement to `tar`. They standardised some things vendors had been doing for a while, but unfortunately (like all POSIX standards) there were several things that were left under-specified. Extended attributes is the most obvious example, as well as how to handle new file-types (other than just doing it the old-fashioned `ustar` way).

The [Extensions] Strike Back

With POSIX.1-2001 and PAX, surely we're all done and there's nothing left for vendors to extend, right? Oh my sweet summer child, if only that were the case. To cut the vendors some slack (especially Jörg Schilling's `star`, where most of the work on sane extensions has happened), PAX simply didn't specify enough things to be

usable as an archive format on modern Unix-like systems. So extensions were necessary, and this time folks weren't limited to just 26 extension slots.

`star` has an enormous number of extensions, many of which I won't get into here because most of them are *exceptionally* niche and you probably aren't interested. But there are a few important ones we should quickly discuss.

Extended attributes are an absolutely awful beast, and `tar` makes it even harder to actually use them. First of all, not all Unix-like systems have the same ideas of what an extended attribute is (since it's – surprise – not defined in POSIX and yet everyone has their own flavour of it). This automatically makes it ludicrously hard to support them in the first place, but then you get into how the support actually turned out – and that's a *whole different* flavour of trash-fire.

There are five different extensions for storing them. The BSDs use `LIBARCHIVE.xattr.<name>=<value>` and `star` uses the very similar `SCHILY.xattr.<name>=<value>` (though only `libarchive` supports binary extended attributes using *HTTP-style* `%`*-encoding*). Apple's `tar` is really out there and uses a special "resource file" with `._` prefixed to the basename of the file in question, which contains some Apple-specific magic that is used to represent extended attributes. AIX `tar` uses the `E` typeflag – because it was added during the pre-PAX days, as does Solaris (though of course, it's done incompatibly). If you're confused, don't worry – so is everyone else.

Another related problem to extended attributes is "POSIX" (it was never in an actual standard) and NFSv4 ACLs. On Linux, NFSv4 ACLs are represented as extended attributes, which is a really fun time. I won't get too far into ACLs, since you rarely run into them. But in short, `star` has a lot of extensions for NFSv4 ACLs and POSIX.1e-2001 ACLs are fairly complicated, to say the least. Again, there are incompatibilities between different implementations.

There are some forms of extended metadata that most people forget exist, like `chattr(1)` "file attributes", which are not even included in most vendor implementations (`star` uses `SCHILY.fflags`). Yet again, this is another case of an extension that wasn't widely supported (GNU `tar` doesn't support this metadata type at all, as far as I can tell).

Another problem that arose out of the current extension hell is that you can have files that use different extensions for the same `tar` entry (not all extensions support everything you might want – so you need to mix-and-match for some cases). This massively increases the complexity of most `tar` implementations (and some implementations like Go's `archive/tar` are still struggling with it).

I could go on with the countless extensions and problems that arise form them, but

I'm sure I'm boring you by now. The key take-away is that these extensions have all resulted in the same interoperability issues as the past, and in quite a few cases vendors re-invent each others' extensions (because they need them, POSIX doesn't provide them, so they end up NIH-ing them).

Where Are We Today?

As a result, these days when you refer to `tar` you are actually referring to a collection of different formats that have been re-implementing each others' extensions slightly differently for decades. And while PAX, `star`, BSD `tar`, and GNU `tar` are all mostly interoperable there are decades worth of legacy powering this whole ship.

Examples of where issues like this crop up are Go's `archive/tar` library that now has a deceptively simple-looking `Format` attribute which allows you to forcefully select a `tar` format to use (if you don't explicitly use GNU or PAX then the `atime` and `ctime` will not be included in the archive). Furthermore, recent Go versions have changed the default output of `archive/tar` in ways that are new readings of the PAX specification. To put it simply, `tar` is what most implementations seem to support (which is usually PAX-plus-extensions) – and that's not a really good bedrock to use for a new standard (as I mentioned before, even agreeing on "what is `tar`" can be difficult).

I will admit that I enjoy using tools that were written long before I was born (since I'm actually a huge critic of almost all NIH projects), but you should ask whether you are reaching for a tool out of familiarity or because you earnestly believe it is the best tool for the job.

But What Practical Issues Are There?

All of this history might be interesting (well, to me at least), but it's hardly a reason to not use a format right? Any old format will have similar growing pains, and given the ubiquity of `tar` it seems fairly drastic to not use it *just* because it's old. Unfortunately there are a whole host of practical problems with `tar` for container images, which can be found by looking at what we might want in a theoretical container image format. Here is the list I have, and will be working through:

- Machine-independent representation.
- De-duplication (both transfer and storage).
- Parallelisable (both transfer and extraction).
- Sane handling of deleted files.
- Reproducible, with a canonical representation.
- Non-avalanching.
- Transparent.

For each of these, I will go into some depth what they mean and how `tar`-based container images cannot provide them to a degree that is satisfactory. It might be a bit of a long ride, but I hope that this will help explain why `tar` is fundamentally not a good match for this problem. It might be possible to modify `tar` beyond recognition, but then the only benefit of `tar` (it's ubiquity) is lost because we've just created an incompatible variant of `tar`. In fact (as we'll see in a minute), container images **already are incompatible variants of** `tar` when you look at how white-outs work!

Machine-Independent Representation

Specifically, it should be possible to create a container image on any given machine and it should work on any other machine. In fact, ideally you would hope that machine-specific configurations shouldn't affect the container image's creation and all machines should be able to equally use the image regardless of their machine-specific configuration. The latter statement is more general and is harder to get.

Arguably, this is something that `tar` was designed for. And so it does quite well here – most machine-specific things (inode numbers, filesystem-specific layout information, and so on) are not embedded into `tar` archives. Similarly, extraction of a `tar` archive is the same regardless of filesystem.

So, we're all good – right? Unfortunately no. While `tar` does quite well here, you can run into a variety of issues very quickly.

First of all, `tar` archive entries can be put in any order and it's still semantically the same `tar` archive. This is a problem for reproducibility but let's deal with that later. In the context of machine-independence, the ordering of a `tar` archive's entries can be impacted by the filesystem used. In particular, the ordering of directory entries in `readdir(3)` is dependent on how the filesystem stores directory entries. Many container image implementations sort them in user-space in an attempt to get around this problem, but most `tar` implementations do not. Thus, to preserve ubiquity we must admit that `tar` can result in this type of change based on host-specific conditions. Extraction is unaffected by this, but it harms reproducibility.

In addition, extended attributes (`xattrs`) are a real pain. Their ordering in

`llistxattr(2)` is also completely filesystem-dependent and will affect how they are ordered in the `tar` archive (not to mention there are several ways of representing them). There are several other problems with `xattrs`, which I will expand on in reproducibility.

All-in-all though, `tar` does pretty well here. Too bad this is the only section where that's the case.

Lack of De-duplication

De-duplication is pretty important for container images, because without it we might as well be shipping around a rootfs archive for the entire image each time (this is actually want LXD does – though with a bit more care).

It should be noted that I've separated de-duplication into two forms, since there is a clear difference between not having to re-download bits that you already have (transfer de-duplication) and saving disk space when the image is on-disk and in-use (storage de-duplication). Ideally our format should help us with both problems, but different users care about one more than the other (depending on what they are optimising for).

To put it bluntly, `tar` archives provide no standard method of de-duplication and in fact almost encourage duplication on every level (and the extensions that add de-duplication won't help us). `tar` archives have no internal de-duplication other than hard-links which are not really a form of de-duplication within our format because they require the on-disk image to be using hard-links.

What we're really talking about here is how `tar` layers operate with regards to de-duplication. And to be honest, `tar` layer-based de-duplication is **effectively useless** outside of the `FROM <foo>` flow of Dockerfiles. Updating a base image requires you to re-download the whole thing and store it entirely separately. If only a single byte in a single package has changed, that's tough – you just have to re-download and store another 50MB. That's just awful, and has resulted in a lot of folks moving to smaller container images (which is a mistake in my opinion – distributions serve a clear purpose and hacking away bits of a distribution image or switching to a niche distribution shouldn't be done lightly).

In addition, there are many places where duplication is rampant:

- If you modify the metadata or bit-flip a large file in a lower layer, the next `tar` layer has to include the entire file contents. `tar` doesn't have metadata or partial-content entries. Solaris had an extension for it called `LF_META`, but see my above rant about extensions. `star` also has a similar (but incompatible) extension using PAX's keywords with `SCHILY.tarfiletype=meta`, and the same rant applies.
- If you delete a file, then a "white-out" needs to be stored in the next layer (which is effectively a tombstone) – meaning that **removing a file increases the size of our image**. As an aside, this tombstone actually means that standard `tar` implementation will not be able to correctly extract a container image (we've already forked from standard `tar`). A very fun restriction added by these tombstones is that you cannot have a container image that has a file containing a `.wh.` prefix. I will go into more detail about white-outs in a later section.
- If you create a hardlink to an existing file in a previous layer, in order for the new layer's `tar` archive to be valid you need to copy the original file into the new `tar` archive as well as add the hardlink entry (`tar` archives have hardlink entries which just store the target of the link). This is fairly expensive duplication (especially if the file is large) and can't really be fixed without generating archives that are no longer valid and self-contained. Hardlinks are also a pretty large pain in the `tar` format anyway, but I won't get into that much here.

And while you do get layer de-duplication because layers are content-addressable, the layers themselves are so fragile (a single bit-flip makes the entire layer hash different) that you end up with very little practical de-duplication (of transfer and storage).

Lack of Parallelisable Operations

Given that our machines have the ability to multi-task, it'd be nice if we weren't bottle-necked on many image operations. In particular, transfer and extraction (taking the image from it's OCI representation and actually putting into a storage driver) are very important to parallelise if possible.

A single `tar` archive cannot be extracted in parallel without a single linear pass (to figure out where the headers start) since `tar` archive entries are header-followed-by-content based. Adding an index might help with this, but requires adding more out-of-spec things to our `tar`-like format. There are some other `tar` forks that have indexes, but as you'll see in a second we'd need something a bit more complicated.

But what about extracting layers in parallel? I'm sure I'm not the only person who has been frustrated that a lot of the time spent on getting a cold-start container to run is in extracting the image. There are a few problems with extracting `tar` layers in parallel (though it actually could be possible to do, it would just be quite difficult without more extensions). Since two layers can contain the same file but with different contents (which means that **the file is extracted twice**), and you have

"white-outs" to deal with (which means that **the file is extracted and immediately deleted**) you can't just extract them all concurrently. You could be more clever about it by extracting them in parallel and making sure that earlier layers don't overwrite later ones. But you're still subject to races (which would decide whether or not you extract the same file more than once) as well as making extraction code quite complicated (figuring out whether two non-existent paths refer to the same file would be a "fun" exercise, as well as dealing with hard-links and the like).

The obvious solution would be to add an index on the whole-image level which tells you what paths are present in each archive (and where their header offset is). There is a slight problem with using the header offset – the PAX extension headers (`x` and `g`) can scattered throughout the archive, and you need to know their values when interpreting a `tar` header. Which means you have to store the whole header once parsed, and then you can use the content offset to extract everything in parallel (since you know which layer has the latest copy of the file). Unfortunately we've just out-sourced the header information to a separate index, and the archives are now just being used as content stores – which means we've invented our own format that uses a stunted form of `tar`. All of this work and gymnastics for no good reason.

In addition, most container images use **compressed** `tar` archives. A compressed archive *cannot be seeked* without extracting everything before it, making partial extraction (or other such partial operations) needlessly expensive. Duplicity hit this problem, and the only way of solving it is to make compression happen underneath the archive format (not above it, as is the case with `tar+gzip`).

Insane Handling of Deleted Files (White-outs are Awful)

This issue is an overlapping of a few other issues such as de-duplication and parallelisation, but is specifically focused on deleted files and white-outs because they deserve extra-special attention.

Because it's possible for a file to be deleted in a layer, it's necessary to be able to represent this in a `tar` archive. And immediately we've hit a barrier – `tar` doesn't support this concept at all. So it's necessary to have some kind of extension for it.

In order to support deleting files in layers, the OCI image format adopted AUFS's on-disk format (the reason for this is historical and is because Docker baked the AUFS on-disk format into their image format since it was the only storage driver they supported originally, and this has been carried into the OCI as legacy). Deleted files (and directories) are represented as an empty regular file (known as a "white-out") with `.wh.` prepended to their `basename`. Aside from being incompatible with other `tar` implementations (which will just extract the weird `.wh.` file without knowing what it is)

it also means that you cannot include a real file with a prefix of `.wh.` inside any OCI image. Personally, I think embedding AUFS's format was a fairly big mistake but we're stuck with it for now. There are other ways of dealing with deleted files, but they all have similar problems with interoperability:

- Using `SCHILY.filetype=white-out` is possibly the best solution, since it's already used by BSD and exists specifically to represent opaque directory entries created through BSD's `mount_unionfs`. However it is a `star`-only extension, and arguably would be somewhat lying about the source of the filesystem being from a BSD `mount_unionfs` (though it would probably interoperate just as well with `star`).
- Using a special entry type that we create ourselves. If an implementation sees our white-out entry type, they will at least have an opportunity to fail loudly (which is somewhat better than the fail-silent `.wh.` approach we have right now). But obviously most implementations won't support our special white-out entry type, breaking interoperability.
- Having an external deleted file list. This is nice because it doesn't require touching the `tar` format, but it comes with the downside that the archives no longer fully describe the root filesystem (and users have to be aware of this because it no longer is just "good old `tar`"). If we have to supplement `tar` to make it work, why still use `tar`?
- Copying the representation that Linux's `overlayfs` uses, which is to use device number `{0,0}` for non-directories and the `xattr` `overlay.opaque=y` for directories. The main problem with this is that it is repeating the AUFS mistake again by baking a particular overlay filesystem's representation of white-outs into a format – as a result it won't be interoperable. Not to mention that if you wanted to store a real `overlayfs` directory inside a container image you wouldn't be able to (because on extraction there would be no way of telling if the white-outs are meant to be inside the image or are the image's own white-outs). In addition, `overlayfs` has changed their white-out format in the past, so baking it into our format seems like a bad idea.
- Creating our own fake `xattr` (like `opencontainers.whiteout=y`) to represent all white-out files. This is potentially better than copying `overlayfs`, as it means we don't need to worry about not being able to represent `overlayfs` directories inside container images. It also doesn't conflict with anything, because we invented it. And (on Linux at least), the `xattr` namespaces are quite restrictive to write to and I don't think you could actually set `opencontainers.` xattrs on any Linux filesystem (though on other operating systems this might be possible – which would lead to issues of not being able to store any universal filesystem structure). The main downside is that we are *explicitly* removing any chance of interoperability without convincing other `tar` implementations to implement our weird format.

All of these are fairly disappointing solutions (though it is nice that `star` has something we could re-use that is at least somewhat interoperable). This is a direct result of trying to have a layered format built with another format that wasn't designed for layering. In addition, layering causes fun problems because the image history is contained in the image. Embedding the history of an image in every image has

caused some security concerns in the past related to having build-time secrets that would be included in layers and attempting to redact them by deleting them didn't remove them from the previous layers (something that is somewhat of a restriction made by using `tar` layers, but is also more of a workflow issue).

Lack of Reproducibility and Canonical Representation

Reproducible builds have gotten quite a lot of hard work put into them in the past few years. These days, a vast majority of the packages available in distributions are built bit-for-bit reproducibly – which is an astonishing achievement (and allows for far more independent verification of binaries). It would be fairly self-defeating if the packaging format we use for containers wasn't also reproducible. Not to mention that reproducible images would mean that two image generators that have never communicated could benefit from de-duplication. You could reproducibly create a distribution image yourself (using the distribution's build scripts and sources), and still be able to de-duplicate with it! A canonical representation is very important in order to make sure that all image generators will always produce reproducible representations of the image (any lee-way will allow for dissonance). Not to mention that you could now verify distribution images in more ways than currently available (such as adding your own extra verifications during image build and verifying that the final image is identical to the distribution one).

Due to a large variety of reasons, `tar` archives are practically impossible to reproduce effectively. While there is no technical reason why they are hard to reproduce, there are a myriad of complications that make it difficult to reproducibly create the same archive. There are projects which attempt to solve this problem, but the fundamental issue remains (and `tar-split` only allows you to take an archive and make it look like another version of itself if you have the pre-generated metadata).

One of the most obvious problems (as mentioned in the machine-independent section) is that you can re-order archive entries without issue. This results in trivially different representations, and without a canonical representation they're all as good as one another (most implementations end up storing them in the order given by the filesystem).

The other really obvious problem is that different extensions overlap significantly, resulting in there being many different ways of representing (with different extensions) things not available with the base `tar` format. Examples include long path names, or new file types. Compounded with no canonical format (and that different `tar` generators and consumers having differing levels of support for mixed extensions in the same archive), you end up with a real mess and the same filesystem having many different representations.

Extended attributes are a really bad problem, on many levels (to the point where I could make a whole blog post just about that). As I mentioned in the history of `tar`, there are **five different extensions for storing them**. This means that an implementation could use any of them and still be a valid `tar` archive – furthering the canonical representation problems.

Then you have how languages treat `xattrs`. Since they're basically a key-value store for metadata almost every library developer thinks they should be stored in hash tables. This means that their iteration order is randomised. Which means that the output archive has random `xattr` order and thus is not reproducible and has no canonical format. Many languages also incorrectly assume that `xattrs` can only contain valid UTF-8 (or ASCII) strings – this is also false, they can contain arbitrary binary data. I have yet to see a tool that handles this correctly. Also empty `xattrs` are entirely valid, but PAX doesn't allow them – so there are valid filesystems that cannot be represented with `tar` (aside from the `.wh.` problem I outlined earlier). How awesome is that!

But my favourite thing is that a given `tar` implementation can start producing different archives between versions, for any variety of reasons. With Go, there were a series of releases where each one changed the default output of the built-in `archive/tar` library. It got so bad I had to add regression tests for the language in umoci. And all of this is possible because there is no defined canonical representation of a `tar` archive and so library developers feel free (quite rightly) that they can change the default output – it's entirely our own fault we're depending on it not to change (there was even a proposal to randomise the output of `archive/tar`).

Avalanching

I've borrowed this term-of-art from cryptography. In this context, it means that a small change in an image results in a disproportionately large change such that we need to re-download or store much more data. This is slightly different from de-duplication in that it's about the way the format handles small changes rather than how we handle similar data throughout all images (though a single solution can solve both issues).

`tar` layers are avalanching by nature, because any change in a layer results in us needing to download the whole thing all over again. I'm not sure it's necessary to elaborate this point, since I went over it in the de-duplication section.

Lack of Transparency

This last one is a fairly nebulous goal, and is one that will require quite a bit of thought. Effectively the problem is that currently almost all techniques for finding

security vulnerabilities is to scan the filesystem. But distributions already know what packages they have (and what security vulnerabilities were fixed in those package versions). So why are we duplicating work – why can't you just have a verifiable manifest of the packages in an image?

In the Open Build Service we have this, though it's only used internally so that OBS knows when to re-build a container image (if any of the packages in the image are updated in the dependency tree). However, this information would be useful to more than just distribution folks – the security status of packages is something that distributions **and distributions alone** know for sure.

In this case, `tar` doesn't make things easier or harder for us. Transparency needs to be added as a manifest on top (though `tar` archives might make verification of the manifest harder, since they need to be extracted in-memory and also the format is opaque to the OCI image archives). With a less opaque format, it might be possible to make it easier to verify that a particular package is verbatim present and that the manifest is complete.

I'll be honest, my current ideas for how to solve this issue are quite primordial (compared to my solutions for the other issues I've listed). I believe my new format idea *could* help make this easier, but it will still require a fair bit of work (ideally a Merkle tree would allow us to combine the filesystem tree of packages and verify that a package is present fairly trivially, but doing so would compromise the canonical representation goal). I'm still thinking on how this particular issue can be solved.

How Do We Get It?

I'm afraid to find that out, you'll need to wait until the next instalment. I hope to get it complete in a few weeks (I was hoping to have a PoC available with the next instalment but that's just a silly goal at this point).

If you want a taste though, the general idea is that we can resolve most of the issues I've listed and gain most of the properties we want by creating our own format that is built on top of the OCI content-addressable store, and is a Merkle tree with content-defined chunking of file contents. The basic idea is very similar to backup tools like restic or borgbackup. Since OCI has smart pointers, we can define a few new media-types, and then our new format would be completely transparent to OCI image tools (as opposed to opaque `tar` archives).

But you'll learn all about that next time. Thanks for reading, and happy hacking!

Unless otherwise stated, all of the opinions in the above post are solely my own and do not necessary represent the views of anyone else. This post is released under the Creative Commons BY-SA 4.0 license.

Want to keep up to date with my posts?
You can subscribe to the Atom Feed.

normal