 292 Votes

Introduction

My first encounter with docker goes back to early 2015. Docker was experimented with to find out whether it could benefit us. At the time it wasn't possible to run a container [in the background] and there wasn't any command to see what was running, debug or ssh into the container. The experiment was quick, Docker was useless and closer to an alpha prototype than a release.

Fast forward to 2016. New job, new company and docker hype is growing like mad. Developers here have pushed docker into production projects, we're stuck with it. On the bright side, the run command finally works, we can start, stop and see containers. It is functional.

We have 12 dockerized applications running in production as we write this article, spread over 31 hosts on AWS (1 docker app per host [note: keep reading to know why]).

The following article narrates our journey with Docker, an adventure full of dangers and unexpected turns.

Production Issues with Docker

Docker Issue: Breaking changes and regressions

We ran all these versions (or tried to):

```
1.6 => 1.7 => 1.8 => 1.9 => 1.10 => 1.11 => 1.12
```

Each new version came with breaking changes. We started on docker 1.6 early this year to run a single application.

We updated 3 months later because we needed a fix only available in later versions. The 1.6 branch was already abandoned.

The versions 1.7 and 1.8 couldn't run. We moved to the 1.9 only to find a critical bug on it two weeks later, so we upgraded (again!) to the 1.10.

There are all kind of subtle regressions between Docker versions. It's constantly breaking unpredictable stuff in unexpected ways.

The most tricky regressions we had to debug were network related. Docker is entirely abstracting the host networking. It's a big mess of port redirection, DNS tricks and virtual networks.

Bonus: Docker was removed from the official Debian repository last year, then the package got renamed from *docker.io* to *docker-engine*. Documentation and resources predating this change are obsolete.

Docker Issue: Can't clean old images

The most requested and most lacking feature in Docker is a command to clean older images (older than X days or not used for X days, whatever). **Space is a critical issue given that images are renewed frequently and they may take more than 1GB each.**

The only way to clean space is to run this hack, preferably in cron every day:

```
docker images -q -a | xargs --no-run-if-empty docker rmi
```

It enumerates all images and remove them. The ones currently in use by running containers cannot be removed (it gives an error). It is dirty but it gets the job done.

The docker journey begins with a clean up script. It is an initiation rite every organization has to go through.

Many attempts can be found on the internet, none of which works well. There is no API to list images with dates, sometimes there are but they are deprecated within 6 months. One common strategy is to read date attribute from image files and call '*docker rmi*' but it fails when the naming changes. Another strategy is to read date attributes and delete files directly but it causes corruption if not done perfectly, and **it cannot be done perfectly except by Docker itself.**

Docker Issue: Kernel support (or lack thereof)

There are endless issues related to the interactions between the kernel, the distribution, docker and the filesystem

We are using Debian stable with backports, in production. We started running on Debian Jessie 3.16.7-ckt20-1 (released November 2015). This one suffers from a major critical bug that crashes hosts erratically (every few hours in average).

Linux 3.x: Unstable storage drivers

Docker has various storage drivers. The only one (allegedly) wildly supported is AUFS.

The AUFS driver is unstable. It suffers from critical bugs provoking kernel panics and corrupting data.

It's broken on [at least] all "linux-3.16.x" kernel. There is no cure.

We follow Debian and kernel updates very closely. Debian published special patches outside the regular cycle. There was one major bugfix to AUFS around March 2016. We thought it was THE TRUE ONE FIX but it turned out that it wasn't. The kernel panics happened less frequently afterwards (every week, instead of every day) but they were still loud and present.

Once during this summer there was a regression among a major update, that brought back a previous critical issue. It started killing CI servers one by one, with 2 hours in average between murders. An emergency patch was quickly released to fix the regression.

There were multiple fixes to AUFS published along the year 2016. Some critical issues were fixed but there are many more still left. **AUFS is unstable on [at least] all "linux-3.16.x" kernels.**

- Debian stable is stuck on kernel 3.16. It's unstable. There is nothing to do about it except switching to Debian testing (which can use the kernel 4).
- Ubuntu LTS is running kernel 3.19. There is no guarantee that this latest update fixes the issue. Changing our main OS would be a major disruption but we were so desperate that we considered it for a while.
- RHEL/CentOS-6 is on kernel 2.x and RHEL/CentoS-7 is on kernel 3.10 (with many later backports done by RedHat).

Linux 4.x: The kernel officially dropped docker support

It is well-known that AUFS has endless issues and it's regarded as dead weight by the developers. As a long-standing goal, **the AUFS filesystem was finally dropped in kernel version 4.**

There is no unofficial patch to support it, there is no optional module, there is no backport whatsoever, nothing. **AUFS is entirely gone**.

[dramatic pause]

.

.

.

How does docker work without AUFS then? Well, it doesn't.

[dramatic pause]

.

.

.

So, the docker guys wrote a new filesystem, called overlay.

"*OverlayFS is a modern union filesystem that is similar to AUFS. In comparison to AUFS, OverlayFS has a simpler design, has been in the mainline Linux kernel since version 3.18 and is potentially faster*." — Docker OverlayFS driver

Note that it's not backported to existing distributions. Docker never cared about [backward] compatibility.

Update after comments: Overlay is the name of both the kernel module to support it (developed by linux maintainers) and the docker storage driver to use it (part of docker, developed by docker). They are two different components [with a possible overlap of history and developers]. The issues seem mostly related to the docker storage driver, not the filesystem itself.

The debacle of Overlay

A filesystem driver is a complex piece of software and it requires a very high level of reliability. The long time readers will remember the Linux migration from ext3 to ext4. It took time to write, more time to debug and an eternity to be shipped as the default filesystem in popular distributions.

Making a new filesystem in 1 year is an impossible mission. It's actually laughable when considering that the task is assigned to Docker, they have a track record of unstability and disastrous breaking changes, exactly what we don't want in a filesystem.

Long story short. That did not go well. You can still find horror stories with Google.

Overlay development was abandoned within 1 year of its initial release.

[dramatic pause]

.

.

.

Then comes Overlay2.

"*The overlay2 driver addresses overlay limitations, but is only compatible with Linux kernel 4.0 [or later] and docker 1.12*" — Overlay vs Overlay2 storage drivers

Making a new filesystem in 1 year is still an impossible mission. Docker just tried and failed. Yet they're trying again! We'll see how it turns out in a few years.

Right now it's not supported on any systems we run. We can't use it, we can't even test it.

Lesson learnt: As you can see with Overlay then Overlay2. No backport. No patch. No retro compatibility. Docker only moves forward and breaks things. If you want to adopt Docker, you'll have to move forward as well, following the releases from docker, the kernel, the distribution, the filesystems and some dependencies.

Bonus: The worldwide docker outage

On 02 June 2016, at approximately 9am (London Time). New repository keys are pushed to the docker public repository.

As a direct consequence, any run of "*apt-get update*" (or equivalent) on a system configured with the broken repo will fail with an error "*Error https://apt.dockerproject.org/ Hash Sum mismatch*"

This issue is worldwide. It affects ALL systems on the planet configured with the docker repository. It is confirmed on all Debian and ubuntu versions, independent of OS and docker versions.

All CI pipelines in the world which rely on docker setup/update or a system setup/update are broken. It is impossible to run a system update or upgrade on an existing system. It's impossible to create a new system and install docker on it.

After a while. We get an update from a docker employee: "*To give an update; I raised this issue internally, but the people needed to fix this are in the San Francisco timezone [8 hours difference with London], so they're not present yet.*"

I personally announce that internally to our developers. Today, there is no Docker CI and we can't create new systems nor update existing systems which have a dependency on docker. All our hope lies on a dude in San Francisco, currently sleeping.

[pause waiting for the fix, that's when free food and drinks come in handy]

An update is posted from a Docker guy in Florida at around 3pm (London Time). He's awake, he's found out the issue and he's working on the fix.

Keys and packages are republished later.

We try and confirm the fix at around 5pm (London Time).

That was a 7 hours interplanetary outage because of Docker. All that's left from the outage is a few messages on a GitHub issue. There was no postmortem. It had little (none?) tech news or press coverage, in spite of the catastrophic failure.

Docker Registry

The docker registry is storing and serving docker images.

```
Automatic CI build  ===> (on success) push the image to ===>
```

```
Deploy command <=== pull the image from <=== docker registry
```

There is a public registry operated by docker. As an organization, we also run our own internal docker registry. It's a docker image running inside docker on a docker host (that's quite meta). The docker registry is the most used docker image.

There are 3 versions of the docker registry. The client can pull indifferently from any:

- The Registry v1, now deprecated and abandoned
- The Registry v2, a full rewrite in Go, first released in April 2015
- The Trusted Registry, a (paid?) service mentioned everywhere in the doc, not sure what it is, just ignore it

Docker Registry Issue: Abandon and Extinguish

The docker registry v2 is as a full rewrite. The registry v1 was retired soon after the v2 release.

We had to install a new thing (again!) just to keep docker working. They changed the configuration, the URLs, the paths, the endpoints.

The transition to the registry v2 was not seamless. We had to fix our setup, our builds and our deploy scripts.

Lesson learnt: Do not trust on any docker tool or API. They are constantly abandoned and extinguished.

One of the goal of the registry v2 is to bring a better API. It's documented here, a documentation that we don't remember existed 9 months ago.

Docker Registry Issue: Can't clean images

It's impossible to remove images from the docker registry. There is no garbage collection either, the doc mentions one but it's not real. (The images do have compression and de-duplication but that's a different matter).

The registry just grows forever. Our registry can grow by 50 GB per week.

We can't have a server with an unlimited amount of storage. Our registry ran out of space a few times, unleashing hell in our build pipeline, then we moved the image storage to S3.

Lesson learnt: Use S3 to store images (it's supported out-of-the-box).

We performed a manual clean-up 3 times in total. In all cases we had to stop the registry, erase all the storage and start a new registry container. (Luckily, we can re-build the latest docker images with our CI).

Lesson learnt: Deleting any file or folder manually from the docker registry storage WILL corrupt it.

To this day, it's not possible to remove an image from the docker registry. There is no API either. (One of the point of the v2 was to have a better API. Mission failed).

Docker Issue: The release cycle

The docker release cycle is the only constant in the Docker ecosystem:

1. Abandon whatever exists
2. Make new stuff and release
3. Ignore existing users and retro compatibility

The release cycle applies but is not limited to: docker versions, features, filesystems, the docker registry, all API…

Judging by the past history of Docker, we can approximate that anything made by Docker has a half-life of about 1 year, meaning that **half of what exist now will be abandoned [and extinguished] in 1 year**. There will usually be a replacement available, that is not fully compatible with what it's supposed to replace, and may or may not run on the same ecosystem (if at all).

"*We make software not for people to use but because we like to make new stuff*."
— Future Docker Epitaph

The current status-quo on Docker in our organization

Growing in web and micro services

Docker first came in through a web application. At the time, it was an easy way for the developers to package and deploy it. They tried it and adopted it quickly. Then it spread to some micro services, as we started to adopt a micro services architecture.

Web applications and micro services are similar. They are stateless applications, they can be started, stopped, killed, restarted without thinking. All the hard stuff is delegated to external systems (databases and backend systems).

The docker adoption started with minor new services. At first, everything worked fine in dev, in testing and in production. The kernel panics slowly began to happen as more web services and web applications were dockerized. The stability issues became more prominent and impactful as we grew.

A few patches and regressions were published over the year. We've been playing catchup & workaround with Docker for a while now. It is a pain but it doesn't seem to discourage people from adopting Docker. Support and demand is still growing inside the organisation.

Note: None of the failures ever affected any customer or funds. We are quite successful at containing Docker.

Banned from the core

We have some critical applications running in Erlang, managed by a few guys in the 'core' team.

They tried to run some of their applications in Docker. It didn't work. For some reasons, Erlang applications and docker didn't go along.

It was done a long time ago and we don't remember all the details. Erlang has particular ideas about how the system/networking should behave and the expected load was in thousands of requests per second. Any unstability or incompatibility could justify an outstanding failure. (We know for sure now that the versions used during the trial suffered from multiple major unstability issues).

The trial raised a red flag. **Docker is not ready for anything critical**. It was the right call. The later crashes and issues managed to confirm it.

We only use Erlang for critical applications. For example, **the core guys are responsible for a payment system that handled $96,544,800 in transaction this month**. It includes a couple of applications and databases, all of which are under their responsibilities.

Docker is a dangerous liability that could put millions at risk. It is banned from all core systems.

Banned from the DBA

Docker is meant to be stateless. Containers have no permanent disk storage, whatever happens is ephemeral and is gone when the container stops. Containers are not meant to store data. Actually, they are meant by design to NOT store data. Any attempt to go against this philosophy is bound to disaster.

Moreover. Docker is locking away processes and files through its abstraction, they are unreachable as if they didn't exist. It prevents from doing any sort of recovery if something goes wrong.

Long story short. **Docker SHALL NOT run databases in production, by design.**

It gets worse than that. Remember the ongoing kernel panics with docker?

A crash would destroy the database and affect all systems connecting to it. It is an erratic bug, triggered more frequently under intensive usage. A database is the ultimate IO intensive load, that's a guaranteed kernel panic. Plus, there is another bug that can corrupt the docker mount (destroying all data) and possibly the system filesystem as well (if they're on the same disk).

Nightmare scenario: The host is crashed and the disk gets corrupted, destroying the host system and all data in the process.

Conclusion: **Docker MUST NOT run any databases in production, EVER.**

Every once in a while, someone will come and ask "*why don't we put these databases into docker*?" and we'll tell some of our numerous war stories, so far, no-one asked twice.

Note: We started going over our Docker history as an integral part of our on boarding process. That's the new damage control philosophy, kill the very idea of docker before it gets any chance to grow and kill us.

A Personal Opinion

Docker is gaining momentum, there is some crazy fanatic support out there. **The docker hype is not only a technological liability any more, it has evolved into a sociological problem as well.**

The perimeter is controlled at the moment, limited to some stateless web applications and micro services. It's unimportant stuff, they can be dockerized and crash once a day, I do not care.

So far, all people who wanted to use docker for important stuff have stopped after a quick discussion. My biggest fear is that one day, a docker fanatic will not listen to reason and keep pushing. I'll be forced to barrage him and it might not be pretty.

Nightmare scenario: The future accounting cluster revamp, currently holding $23M in customer funds (the M is for million dollars). There is already one guy who genuinely asked the architect "*why don't you put these databases into docker?*", there is no word to describe the face of the architect.

My duty is to customers. Protecting them and their money.

Surviving Docker in Production

Follow releases and change logs

Track versions and change logs closely for kernel, OS, distributions, docker and everything in between. Look for bugs, hope for patches, read everything with attention.

```
ansible '*' -m shell -a "uname -a"
```

Let docker crash

Let docker crash. self-explanatory.

Once in a while, we look at which servers are dead and we force reboot them.

Have 3 instances of everything

High availability require to have at least 2 instances per service, to survive one instance failure.

When using docker for anything remotely important, we should have 3 instances of it. Docker die all the time, we need a margin of error to support 2 crashes in a raw to the same service.

Most of the time, it's CI or test instances that crash. (They run lots of intensive tests, the issues are particularly outstanding). We've got a lot of these. Sometimes there are 3 of them crashing in a row in an afternoon.

Don't put data in Docker

Services which store data cannot be dockerized.

Docker is designed to NOT store data. Don't go against it, it's a recipe for disaster.

On top, there are current issues killing the server and potentially destroying the data so that's really a big no-go.

Don't run anything important in Docker

Docker WILL crash. Docker WILL destroy everything it touches.

It must be limited to applications which can crash without causing downtime. That means mostly stateless applications, that can just be restarted somewhere else.

Put docker in auto scaling groups

Docker applications should be run in auto-scaling groups. (Note: We're not fully there yet).

Whenever an instance is crashed, it's automatically replaced within 5 minutes. No manual action required. Self healing.

Future roadmap

Docker

The impossible challenge with Docker is to come with a working combination of kernel + distribution + docker version + filesystem.

Right now. We don't know of ANY combination that is stable (Maybe there isn't any?). We actively look for one, constantly testing new systems and patches.

Goal: Find a stable ecosystem to run docker.

It takes 5 years to make a good and stable software, Docker v1.0 is only 28 months old, it didn't have time to mature.

The hardware renewal cycle is 3 years, the distribution release cycle is 18-36 months. Docker didn't exist in the previous cycle so systems couldn't consider compatibility with it. To make matters worse, it depends on many advanced system internals that are relatively new and didn't have time to mature either, nor reach the distributions.

That could be a decent software in 5 years. Wait and see.

Goal: Wait for things to get better. Try to not go bankrupt in the meantime.

Use auto scaling groups

Docker is limited to stateless applications. If an application can be packaged as a Docker Image, it can be packaged as an AMI. If an application can run in Docker, it can run in an auto scaling group.

Most people ignore it but Docker is useless on AWS and it is actually a step back.

First, the point of containers is to save resources by running many containers on the same [big] host. (Let's ignore for a minute the current docker bug that is crashing the host [and all running containers on it], forcing us to run only 1 container per host for reliability).

Thus containers are useless on cloud providers. There is always an instance of the right size. Just create one with appropriate memory/CPU for the application. (The minimum on AWS is t2.nano which is $5 per month for 512MB and 5% of a CPU).

Second, the biggest gain of containers is when there is a complete orchestration system around them to automatically manage creation/stop/start/rolling-update/canary-release/blue-green-deployment. The orchestration systems to achieve that currently do not exist. (That's where Nomad/Mesos/Kubernetes will eventually come in, there are not good enough in their present state).

AWS has auto scaling groups to manage the orchestration and life cycle of instances. It's a tool completely unrelated to the Docker ecosystem yet it can achieve a better result with none of the drawbacks and fuck-ups.

Create an auto-scaling group per service and build an AMI per version (tip: use Packer to build AMI). People are already familiar with managing AMI and instances if operations are on AWS, there isn't much more to learn and there is no trap. The resulting deployment is golden and fully automated. **A setup with auto scaling groups is 3 years ahead of the Docker ecosystem.**

Goal: Put docker services in auto scaling groups to have failures automatically handled.

CoreOS

Update after comments: Docker and CoreOS are made by separate companies.

To give some slack to Docker for once, it requires and depends on a lot of new advanced system internals. A classic distribution cannot upgrade system internals outside of major releases, even if it wanted to.

It makes sense for docker to have (or be?) a special purpose OS with an appropriate update cycle. It may be the only way to have a working bundle of kernel and operating system able to run Docker.

Goal: Trial the CoreOS ecosystem and assess stability.

In the grand scheme of operations, it's doable to separate servers for running containers (on CoreOS) from normal servers (on Debian). Containers are not supposed to know (or care) about what operating systems they are running.

The hassle will be to manage the new OS family (setup, provisioning, upgrade, user accounts, logging, monitoring). No clue how we'll do that or how much work it might be.

Goal: Deploy CoreOS at large.

Kubernetes

One of the [future] major breakthrough is the ability to manage fleets of containers abstracted away from the machines they end up running on, with automatic start/stop/rolling-update and capacity adjustment,

The issue with Docker is that it doesn't do any of that. It's just a dumb container system. It has the drawbacks of containers without the benefits.

There are currently no good, battle tested, production ready orchestration system in existence.

- Mesos is not meant for Docker
- Docker Swarm is not trustworthy
- Nomad has only the most basic features
- Kubernetes is new and experimental

Kubernetes is the only project that intends to solve the hard problems [around containers]. It is backed by resources that none of the other projects have (i.e. Google have a long experience of running containers at scale, they have Googley amount of resources at their disposal and they know how to write working software).

Right now, Kubernetes is young & experimental and it's lacking documentation. The barrier to entry is painful and it's far from perfection. Nonetheless, it is [somewhat] working and already benefiting a handful of people.

In the long-term, Kubernetes is the future. It's a major breakthrough (or to be accurate, it's the final brick that is missing for containers to be a major [r]evolution in infrastructure management).

The question is not whether to adopt Kubernetes, the question is when to adopt it?

Goal: Keep an eye on Kubernetes.

Note: Kubernetes needs docker to run. It's gonna be affected by all docker issues. (For example, do not try Kubernetes on anything else than CoreOS).

Google Cloud: Google Container Engine

As we said before, there is no known stable combination of OS + kernel + distribution + docker version, thus there is no stable ecosystem to run Kubernetes on. That's a problem.

There is a potential workaround: Google Container Engine. It is a hosted Kubernetes (and Docker) as a service, part of Google Cloud.

Google gotta solve the Docker issues to offer what they are offering, there is no alternative. Incidentally, they might be the only guys who can find a stable ecosystem around Docker, fix the bugs, and sell that ready-to-use as a cloud managed service. We might have a shared goal for once.

They already offer the service so that should mean that they already worked around the Docker issues. Thus **the simplest way to have containers working in production (or at-all) may be to use Google Container Engine.**

Goal: Move to Google Cloud, starting with our subsidiaries not locked in on AWS. Ignore the rest of the roadmap as it's made irrelevant.

Google Container Engine: One more reason why Google Cloud is the future and AWS is the past (on top of 33% cheaper instances with 3 times the network speed and IOPS, in average).

Why docker is not yet succeeding in production, July 2015, from the Lead Production Engineer at Shopify.

Docker is not ready for primetime, August 2016.

Docker in Production: A retort, November 2016, a response to this article.

How to deploy an application with Docker… and without Docker, An introduction to application deployment, The HFT Guy.

Disclaimer (please read before you comment)

A bit of context missing from the article. **We are a small shop with a few hundreds servers. At core, we're running a financial system moving around multi-million dollars per day (or billions per year)**.

It's fair to say that we have higher expectations than average and we take production issues rather (too?) seriously.

Overall, it's "*normal*" that you didn't experience all of these issues if you're not using docker at scale in production and/or if you didn't use it for long.

I'd like to point out that these are issues and workarounds happening over a period of [more than] a year, summarized all together in a 10 minutes read. It does amplify the dramatic and painful aspect.

Anyway, whatever happened in the past is already in the past. The most important section is the Roadmap. That's what you need to know to run Docker (or use auto scaling groups instead).