DockAlt: An Alternative to Docker

Katherine Bell
UCLA

Abstract

This report explores an alternative implementation to Docker, a platform that allows software applications to be packaged into portable units. Multiple languages are considered for this implementation. The few described in the pages that follows are: Java, OCaml and Coconut.

1. Introduction

Docker wraps applications in a complete file system that contains all needed components to run that application in isolation: code, runtime, system tools, libraries and anything else that might be needed on a server. These file systems are known as containers. The advantage to this container system is that they allow applications to be run anywhere, regardless of environment. While the implications of this technology are vast (portability, distributed systems, security, etc.), there is concern with Docker's relatively new status as a platform. Additionally, Docker operates from a single source, which implies certain brittleness in its operation.

As such, it has been decided that a second implementation is in order—one that is built on a different programming language. Three languages are being discussed as potential contenders: Java, OCaml and Coconut. In the following pages, I will discuss Docker as a platform in more detail and provide a succinct summary of why the development team behind it decided to go with Go. This will lead to a more informed discussion of the pros and cons of the three aforementioned languages.

2. Docker

Docker is a container technology that makes it easy to package and ship software. It takes a separation of concerns approach, allowing developers to bundle together custom parts for their software applications and run them in isolation on any machine. Docker is built on top of facilities provided by the Linux Kernel. Unlike a virtual machine, Docker containers do not include an operating system. Instead, it relies directly on the host system's own operating system and its underlying infrastructure. Docker has written its own library for interfacing directly the Linux Kernel, which offers low-level access to the operating system. The goal of working directly with the Linux Kernel is to allow the creation of an environment as close as possible to a standard Linux installation without the need for a separate kernel.

Built in this way, Docker is able to maintain a lightweight container system, which starts instantly and uses less RAM than a virtual machine. But Docker also offers a significant expansion on top of the Linux Kernel's capabilities. A few of these features include: versioning, component reuse via the creation of base containers and an expanding repository of shared libraries. Docker is also based on open standards enabling containers to run on all major Linux distributions as well as windows systems and on top of any infrastructure. In short, Docker lets you run more applications in more places with less overhead.

2.1 Why Go?

The developers of Docker chose to develop their product using Go, a relatively new programming language. Their decision was based on a few guiding principles, the most pragmatic of which is Go's language agnostic nature. It's not Java. It's not C++. It's not Python. And it's not PHP. This allowed the Go team to start developing from scratch and without the requisite discomfort arising from language disagreements among its developers.

From a more technical point of view, Go provides static compiling, duck typing, handy tooling, garbage collection, and access to low level APIs,

among other things. Generally speaking, duck typing is thought of as a dynamic approach to typing. However, Go offers a slightly different implementation. Its version of duck typing is better known as "structural typing" as it is checked at compile time. This compromise offers much of the flexibility of type inference that traditional duck typing is known for with far less of the reliability risk. Vis a Vis tooling, Go has built in functionality for running unit tests ('go test') and formatting ('go format'), as well as package management. And perhaps most importantly given Docker's functionality, LinuX Containers (or LXC), which offer an interface to the Linux Kernel, provides a Go API. While Docker currently uses its own native library to work on top of the Linux Kernel, this is a relatively new development and prior to 2014 the LXC API was used.

While the developers of Go cite a few downsides to their choice (no IDE, error handling can be verbose, etc), it is clear that any language chosen as its replacement would have to satisfy Go's same balance of developer-friendliness and reliability, as well as access to LXC or a suitable workaround. Without the ability of the containers to access the underlying operating system and infrastructure much of what makes Docker such an exciting and useful technology would be lost.

3. Java

At first glance Java offers an attractive alternative to Go. It can match Go in terms of reliability and performance. It is statically typed, which lowers the risk of runtime error and compiles to byte code before being run in the JVM (Java Virtual Machine). Java is a well-established language with endless support channels and documentation. It has an arguably larger standard library and set of data types than Go, which is a primary reason Go was chosen. Furthermore, like Docker, one of Java's original intentions as a programming language was that of portability. Its compile once, run everywhere mantra and capability would be useful for the development of a Docker-like platform.

However, Java is a rather obtuse language with a steep learning curve. Unlike the developer-friendly Go, the transition to Java would be difficult provided the current engineering team wasn't already well versed in its methodology. More importantly, Linux offers no Java API to its kernel system. Again, though Docker has now built its own library, for many years it relied on the LXC (LinuX Containers) API. This suggests the development of such a custom library was very cost intensive in terms of developer time and resources. Moreover, because Java uses the JVM and is compiled to byte code instead of machine code, it's really not designed for interfacing with the lowest level of a computer organization. While it would be possible to create a solution using a third party library or a binding wrapped around an implementation in a different language, this would only complicate our platform and add unnecessary maintenance and reliability issues. All in all, this makes Java an unlikely choice for our project as it fails in the one aspect most salient to Docker's construction: it's ability to work with the low-level infrastructure of its host machine.

4. OCaml

Like Java, OCaml is a statically typed language with a robust garbage collector. Programs written in OCaml tend to be clear and succinct given its functional style. Like Go, it also makes use of type inference to speed up the development cycle and place fewer constraints at compile time. Furthermore, as a functional language and the general referential transparency that implies, OCaml easily adapts to the kind of parallelism required to manage multiple containers on a single machine.

However, OCaml is extremely difficult to debug. Its error messages, especially those concerning type, are at best difficult to discern. As such, developers spend an inordinate amount of time working on problems that would be much quicker solved in a different language. This reason alone would make OCaml a suspect choice for developing an entire industry product. Also, as in the case of Java, there is no OCaml API for the Linux Container system indicating that a custom library would have to be built to gain the sort of interaction Docker relies upon. However, there are a number of Github repositories suggesting this kind of implementation would be less cumbersome then the equivalent in Java. Finally OCaml lacks a few basic functionalities, like 32-bit floats for example, that would need to be supplanted by workarounds. Making up for basics such as this could lead to a bloated implementation of DocAlt.

4. Coconut

Coconut is a functional programming language that compiles to Python. Since all valid Python is valid Coconut, and Docker's original implementation (dotCloud) was written in Python, porting Docker's functionality to Coconut should be relatively seamless, at least compared to the other languages discussed thus far. Coconut's syntax is highly readable like that of its parent language, which means the learning curve with this approach would be minimal. More importantly, as a decedent of Python, Coconut is also able to make use of the available LXC API for Python. This puts Coconut at a serious advantage as it completely eliminates one of the major requirements of our Docker implementation. Furthermore, given Coconut's primarily functional style, you gain features like pattern matching and lazy evaluation, which are missing from Python itself. Similar to OCaml, Coconut's functional tendencies could also be useful in constructing a parallelized system for routing container functionality.

But Coconut is brand new. It lacks the support and documentation of a more veteran language. While all python works with Coconut and therefore has access to all of Python's libraries and modules, it begs the question whether or not adding the Coconut layer is really any kind of value add. Writing a new product based on a new language is a risky maneuver. There is no guarantee that the language will be supported in the future.

5. Conclusion

It is tempting to conclude that Coconut is our best option. Inheriting from Python, it has the best Linux support and a potentially invaluable engineering roadmap left behind by the developers of dotCloud. However, the risk of using such a new language shouldn't be overlooked. In order to be able to truly say whether or not this risk is worth it a further cost analysis is needed. Specifically, we should understand how easy it would be to port the Coconut code to pure python if that became necessary. If we plan on developing features based on Coconut's functional capabilities, would going to Python mean significant architectural changes? These are the kinds of questions we have to ask ourselves before making this kind of decision. Still of the three I believe Coconut to be the most promosing.

5. References

https://linuxcontainers.org/lxc/

https://www.upguard.com/articles/docker-vs-lxc

https://www.docker.com/what-docker

https://golang.org/doc/faq#principles

http://theburningmonk.com/2015/05/why-i-like-golang-interfaces/

https://engineeredweb.com/blog/2013/why-go-excellent-programming-language/

http://www.slideshare.net/jpetazzo/docker-and-go-why-did-we-decide-to-write-docker-in-go/40

https://prezi.com/68ivntx5embn/ocaml-pros-and-cons/

http://www.infoworld.com/article/3088058/application-development/python-gains-functional-programming-syntax-via-coconut.html