Containerization Support Languages

Shaan Mathur, *UCLA Computer Science Undergraduate*

Abstract

To best develop a substitute for Docker, we need to to choose a programming language that satisfies several requirements, where our choices have been narrowed down to Java, OCaml, and Scala. To evaluate the languages, we consider their expressiveness in syntax, their support for asynchronous programming, and performance related aspects. After evaluation, it seems that Scala offers us object orientation, functional programming, and high level of expressiveness that makes it a suitable technology to contain multiple applications.

1. Development Considerations

In designing a new version of Docker, one primary consideration we should be considering is the amount of time it takes to develop this new software.

1.1 Syntactic Considerations

Java is a language well known for being fairly verbose. Firstly, every module must be wrapped in a class, so there is no notion of "free floating" functions or methods in the global namespace. This means just to begin writing code, we need to occupy at minimum five lines of code, even for the simplest of tasks. As a unit of metric, we show three implementations of a trivial Process class in the Appendix.

Java's source code seems to be much longer than the others. What is it about Java that makes class design so verbose? For one, all variables need to be declared before using them, so two lines are dedicated just to declaring the instance variables. Then the constructor needs to be created and the parameters have to be assigned to instance variables. Now good design insists that we engage in information hiding via access specifiers, because one day we may want to change the logic of how we access our instance variables and mutate them; therefore, we create wrappers for their access and mutation. Finally we need actual behavior for our class, which involves yet another set of methods.

Though Java's verbosity creates strong structure in our code, it also makes it a bit more unpleasant to read. One frustrating aspect of writing Java code is the need to keep all instance variables private or protected and then specify how to access them via indirection using wrappers. These wrappers are simply being created to make sure that a change of implementation won't break interface; OCaml's approach to this problem is to make the syntax a bit cleaner. As can be seen in the example code for OCaml, we can have in one line an accessor/mutator method, where typical Java form would require three lines. Still, this is not much of an improvement, although there are fewer syntactic structures at play now.

As a programmer maintaining code, why do we need to declare variables outright? Any programmer can logically see that if a variable is being used as an instance variable, then it in fact is an instance variable. If we examine Scala's implementation of the same class, we see that there is no need for declaring instance variables. As a further improvement, all instance variables are by default public. If one ever wanted to change implementation, there is no danger in breaking interface. All one needs to do is simply define special methods that will act as accessor or mutators for the instance variables without having to change any code using the interface. The takeaway is that Scala will grow in verbosity only if it needs to. It is as verbose as it needs to be, which is a huge win in regards to developing quickly and having short, maintainable code.

1.2 Language Design for Asynchrony

One feature that the original designers of Docker strived for was having strong asynchronous primitives [1]. This specifically was needed for requirements such as waiting for I/O or processes. Therefore it is important that the language chosen would have support for asynchronous behavior if it is required.

Java is inherently a synchronous programming language, but it does feature strong support for multithreading as well as a wide array of abstractions for synchronization primitives. Threads can be created and provided a lambda expression in Java 8 to very easily launch an asynchronous method:

```
new Thread( () -> { /* code */} ).start();
```

Therefore it does seem easy to have threads run asynchronously in the background when we just want a task done but do want to wait for its completion.

OCaml also has library support for asynchronous programming in the form of the Async library, which provides primitives like 'Deferred' which is a type that wraps around the actual return type like a future or promise. The language itself also has some inbuilt syntactic sugar meant specifically for asynchronous programming such as >>= which is the equivalent of an 'await' on a Deferred [2]. The library also comes with I/O and networking support as well, which satisfies some of the original developers requirements. OCaml does not have any proper parallelism/multithreading support, however, which means that the language is not flexible enough for future development. After all, if our container is to support concurrent processes, it would be good to use the CPU as efficiently as possible, and this would be hindered if we could not achieve true parallelism.

Scala features its own version of futures with an asynchronous programming library. However Scala's version requires callbacks to invoke in case of success or failure [3]. It also has its own version of true parallelism via threads, so it goes a step beyond OCaml in that regards. Once again, we see that Scala has the power of Java but the succinctness of OCaml, still making it seem like a good choice.

2. Performance

Also of great importance is what the performance of the container will be given the different languages we use. Since we are emulating a lot of OS specific behavior, we need to be sure the language we use will allow us the most efficient use of our resources.

2.1 Compiled vs. Interpreted

One notable difference between the languages we've examined is whether they are compiled and/or interpreted. OCaml, for instance, is a compiled language and so can run natively on most architectures; on the other hand, Java and Scala both run on the JVM and are compiled into bytecode. At this point, however, the JVM's performance has been vastly improving due to Just-In-Time compilers, which will compile code natively if that section of code seems to be exercised a lot.

We also should not be so quick to discount the efficacy of the JVM here. Although OCaml is natively compiled, this means it is less portable then something built with Java or Scala. Although the JVM is in some sense a level of indirection, that indirection provides a portability for any program. This in fact aligns with the whole purpose of the container: to provide a portable interface for applications to run on top of. The container service needs to be portable as well, and so any small sacrifice in performance can be amortized by the portability provided by running on top of the JVM.

Further, if we decide at some point that the JVM is one level of indirection too many, there are native compilers for Java and Scala (though they are third party). So we would at least have the flexibility of choosing what we would like to do later down the road.

2.2 Memory Management

All the languages we have addressed are managed via a garbage collector. The advantage of this feature is that we now do not have to worry about managing dynamically allocated memory, and thus we will essentially be writing an OS that does not suffer from memory leaks or any dangling pointers that would crash the system in kernel mode.

However the garbage collector incurs a heavy performance penalty once a program has been running for a quite a while, and since a container application will be running below other applications of unspecified duration, we need to be flexible enough to handle these scenarios. Our container cannot block other applications because of garbage collection that we are responsible for.

It is also important to realize that because our application is *long-lived*, garbage collection will in fact be delayed for many objects that we use; combine that with the fact that the process is long running and it becomes evident that the garbage collector will have to run several times.

Thus it is important to be able to manage the garbage collector itself to make sure we collect only at optimal times. For instance, we should try to have the garbage collection invoked when the application we are running is idle or closed, or just before we boot up. We should also prioritize garbage collection during periods where only a few applications are running versus many so we only hurt the response time of a few processes.

Java, OCaml, and Scala both provide mechanisms for invoking the JVM garbage collector, so all of these languages do allow for this level of tuning. There are also many variants of the JVM to which one can choose a garbage collector implementation to their liking. OCaml only has one standard and so is inflexible on that front [4].

3. Verdict

Given what we have learned about our three programming languages, we need to decide which is best fit for our task of creating a container.

To build a successful container to host multiple applications, we need a language that will allow us to develop succinct, maintainable code that is extensible. Further, we also need to make sure our software meets particular performance requirements so that it interferes with the application to as minimal a degree as possible.

If we consider using OCaml, we see a few difficulties. The syntactic form is quite succinct, and it does feature the power of object orientation as well as functional programming. It also is natively compiled, meaning we can achieve native speeds in regards to performance. However, our application requires effective synchronization libraries, and the lack of support for real multithreading inhibits us from using OCaml as an effective platform.

Java is a very robust language that is expressive and powerful. It also has a very large set of multithreading support for both true parallelism, concurrency, and asynchronous programming. The JVM even helps because it makes our application portable. It has all the requirements we need for a successful container. However, Java is a very verbose language, and a large piece of software running with Java is going to make our code base larger, which correlates with a decrease in maintainability.

Scala provides all the benefits that Java has but now with the added benefit of a highly expressive and succinct language. We also get to reap some of the functional benefits of OCaml as well, allowing us to have a multi-paradigm language that is succinct, memory managed, and with powerful concurrency primitives. Because of this, we implore that we employ the use of Scala to create the containerization software.

References

[1] M. Crosby, Jerome Petazzoni, Victor Vieux. (November 2013) *Why did we decide to write docker in go?*, Google Developers Group Meetup, Retrieved December 6, 2017.

[2] *Chapter 18. Concurrent Programming with Async* [Project Website], Retrieved December 6, 2017.

[3] P. Haller, A. Prokopec, H. Miller, R. Kuhn, V. Jovanovic. *Futures and Promises*, Retrieved December 6, 2017.

[4] *Garbage Collection - OCaml* [Project Website], Retrieved December 7, 2017.

Appendix

Java

```
public class Process {
        private int pid;
        private long ip;
        public Process(int pid, long ip) {
                this.pid = pid;
                this.ip = ip;
        }
        public int getPID() {
                return this.pid;
        }
        public setPID(int pid) {
                this.pid = pid;
        }
        public getIP() {
                return this.ip;
        }
        public setIP(long ip) {
                this.ip = ip;
        }
        public start() {
                ...
        }
}
```

OCaml

```
class process pid pi =
        object
                val mutable _pid = pid
                val mutable _ip = ip
                method get_pid = _pid
                method get_ip = _ip
                method set_pid id =  _pid <- id
                method set_ip ip = _ip <- ip
                method start =
                        ...
        End
```

Scala

```
class Process(var pid: int, var ip: long) {
        def start(): Unit = {
                ...
        }
}
```