Technology Review

Because of the rather simple nature of the project in terms of its components (a web server and service which acts as a client), all technology can be JVM-based. There is a requirement that Apache Tika be used for one component of the system, but the rest is fair game.

# The Server

There are numerous technologies available for writing a web server, and three languages which can easily facilitate this. Choices:

**Java: Jetty/JAX-RS** - Two mainstream web service technologies for Java.

**Clojure: Ring/Compojure** - The de-facto technology stack for web services in Clojure; built atop Jetty.

**Scala: Jetty/Akka/Spray** - Java Servlet container with middleware and HTTP server libraries.

**Choice**: Clojure with Ring and Compojure**.** Scala has a reputation for having too steep a learning curve to get things going, and this project demands rapid prototyping and eventual development. Java for web servers is simply too verbose and stateful, which makes concurrency particularly difficult - many, many tests would need to be written to ensure correctness. Clojure is a language designed specifically for web servers and concurrency, and its inherent simplicity makes setting up a scalable web service very easy. A simple demo was written (easily) to prove this.

# The Client

As this client is sitting on a web server, the only requirement is that it runs on the JVM. The three languages for the server come into play again. Because of the nature of client code being much more flexible, there is no set of libraries needed to facilitate calling the existing PO.DAAC web service; each language contains constructs necessary to call a web server and read the contents it sends back.

**Choice:** Clojure, with Java baked in if necessary. The “C” in Clojure might as well stand for “Concurrent”, because that was one of the primary goals at design time. Every function in Clojure is capable of being trivially multithreaded or called into by threads because they implement the proper interfaces in Java. Support for sharing data between threads in both an asynchronous and synchronous manner (facilitated through Agents and Atoms, respectively) is baked in an optimized. Core data structures are immutable by default, meaning concurrent boilerplate is not a concern. This also simplifies testing significantly as behavior is always dependable. Because the language was written to interop with Java as needed, it is also a plan to implement code in Java in places where it may be awkward to use Clojure. These times will be apparent as they come, so we aim to be flexible.

# Translation

The core of the application, using Apache Tika to translate Oceanography data and metadata into a target language, is arguably the most important piece. Because Tika is simple a Java library, it can again be used by any language that runs on the JVM.

**Choice:** Clojure, with Java baked in as necessary. The simplicity of Clojure, especially from a testing perspective, makes it a better choice than anything else. Because of the test-heavy nature of making sure data is translated properly, this choice comes down to how easily vast swaths of test code can be written and run at a moment’s notice. Clojure is a functional language, and tests are quite functional in nature - lots of tests can be cranked out very, very easily. There is room for movement, however. Just as with the server, if pieces work best when written in Java, they will be. As a bonus, there exists a Clojure library known as *pantomime* which uses Tika under the covers to detect MIME types - this could simplify development significantly if proven to be useful.