iPReS - Requirements

**Team Name:** Team Warren Buffet

**Team Members:**

Phillip Carter ([carterp@onid.oregonstate.edu](mailto:carterp@onid.oregonstate.edu))

Bhavik Patel ([patelbh@onid.oregonstate.edu](mailto:patelbh@onid.oregonstate.edu))

Daniel Song ([songh@onid.oregonstate.edu](mailto:songh@onid.oregonstate.edu))

**Clients:**

Lewis McGibbney ([lewis.j.mcgibbney@jpl.nasa.gov](mailto:lewis.j.mcgibbney@jpl.nasa.gov))

Chris Mattmann ([chris.a.matmann@nasa.gov](mailto:chris.a.matmann@nasa.gov))

Kathryn O-Brien ([kathryn.obrien@cloudera.com](mailto:kathryn.obrien@cloudera.com))

**Introduction**

The NASA Jet Propulsion Laboratories (JPL) Physical Oceanography Distributed Active Archive Center (PO.DAAC) is one of a number of NASA data archive’s containing many petabytes of oceanographic data. The primary goal (and challenge) for PO.DAAC is to enable provision, dissemination and availability of such data to the global scientific community at large. The driving justification behind the iPReS project is to address the growing requirement for PO.DAAC to provide high quality data products and services in a user-oriented manner by introducing language translation support for any data products retrieved from the data archive. Currently, this information is available only in English.

**Project Description**

There is a strong desire from within the scientific community to have internationalized data products - scientific information and its metadata available in a multitude of languages. PO.DAAC offers a number of development tools within a sandbox-type environment. One of these, the web services API and REST service endpoint exposes a significant amount of oceanographic data and metadata for free consumption. A current limitation of the web service is the inability to provide data products in target languages other than English. The aim for this project is to allow scientists across the world the ability to make use of this data in their native language.

The completed project will be a web service that compliments the existing PO.DAAC web service, querying untranslated data and metadata, translating it into a target language, then exposing the translated information in a desired format on-demand. This will involve three layers: a service-facing client which reads PO.DAAC data, a translation layer which translates data and metadata into a target language, and a client-facing service exposing the translated data.

When completed, an end-to-end service running on JPL’s servers will allow for any scientist to query PO.DAAC oceanography data and receive it in any of the multitude of supported languages.

**Requirements**

1. The system shall ideally support the following languages:

Arabic, Bulgarian, Catalan, Chinese Simplified, Chinese Traditional, Czech, Danish, Dutch, Estonian, Finnish, French, German, Greek, Haitian Creole, Hebrew, Hindi, Hmong Daw, Hungarian, Indonesian, Italian, Japanese, Klingon, Klingon (pIqaD), Korean, Latvian, Lithuanian, Malay, Maltese, Norwegian, Persian, Polish, Portuguese, Romanian, Russian, Slovak, Slovenian, Spanish, Swedish, Thai, Turkish, Ukrainian, Urdu, Vietnamese and Welsh.

1. The system shall take locale into account when supporting languages. **Optional/TBD**.
2. The system shall be capable of handling multiple clients. At the minimum, data requested shall be serviced to the correct client under concurrent request scenarios.
3. The system shall be able to handle at least 25 concurrent requests without experiencing some kind of error.
4. The system shall contain extensive tests to verify Requirements 3 and 4.
5. The system shall be able to break down each piece of a request, and understand which portions of PO.DAAC data sets are being requested.
6. The system shall be able to filter and merge PO.DAAC data sets based on request information from Requirement 6.
7. The system shall contain extensive tests to verify Requirement 7.
8. The system shall contain an Integration Test which verifies the pieces of the system specified by Requirements 3, 4, 6, and 7 communicate with each other correctly.
9. The system shall make use of Apache Tika to translate PO.DAAC data and metadata into a target language.
10. If necessary or convenient, the system shall use Apache Tika to extract metadata from PO.DAAC data sets. **Optional/TBD.**
11. The system shall contain extensive Unit tests to verify Requirement 10.
12. The system shall contain an Integration Test which verifies the pieces of the system specified by Requirements 6, 7, 9, and 10 communicate with each other correctly.
13. The system shall contain a Distributed Cache on the PO.DAAC-facing layer which caches data and metadata which has already been translated. This will be based on popularity, if requested data is a subset of other data sets, etc.
14. The system shall recognize which data sets are to be cached, or which information is already cached (this information is still pending due to the need to communicate with the scientific community about their desires).
15. The system shall contain extensive Unit Tests for Requirements 14 and 15.
16. The system shall be capable of merging translated data sets from the cache and PO.DAAC/Tika services based on client requests.
17. The system shall contain extensive Unit Tests for Requirement 17.
18. The system shall contain an Integration Test which verifies the pieces of the system specified by Requirements 6, 7, 9, 10, 13, 14, and 15 communicate correctly.
19. The system shall make use of Compojure for handling API routing.
20. The system shall make use of Ring, a mature HTTP abstraction library for writing web servers.
21. The system shall use Leiningen (preferred) or Maven for build automation.
22. The client-facing layer shall be capable of exposing requested data sets in Scientific Data Formats Oceanographers are used to using (HDF4, HDF5, etc). The specific data formats will be determined in the future based on the demand for the system.
23. The system shall have extensive tests to ensure information gets exposed in the correct data formats, as requested by a user.
24. The system shall expose information in a variety of non-scientific file formats, such as CSV, PDF, etc. **Optional/TBD.**
25. The system shall aim to tie in with other software tools available within the PO.DAAC development components and sandbox. The aim is to further increase the usefulness of the iPReS service.
26. The system shall adhere to REST methodology, and aim for being easily-integrated into future mobile applications focused on data analysis.
27. The system shall contain a System Load Test which attempts to crash the service when all pieces are connected.
28. The system shall contain extensive tests to verify Requirement 6.

**Versions**

1.0 - Service and Client successfully built to read and expose untampered data - one simple PO.DAAC query supported.

1.1 - The same PO.DAAC query from 1.0, but fully-translated using Apache Tika

1.2 - A basic distributed cache established to support the query from 1.0 and 1.1.

1.3 - A new PO.DAAC query whose results are cached in the distributed cache established above.

1.4 - Extracting information both from the cache and PO.DAAC, merging results. Done via a new query which requires both components.

1.5 - Localization support

1.6.1 - One new language supported

1.6.2 - One new language supported

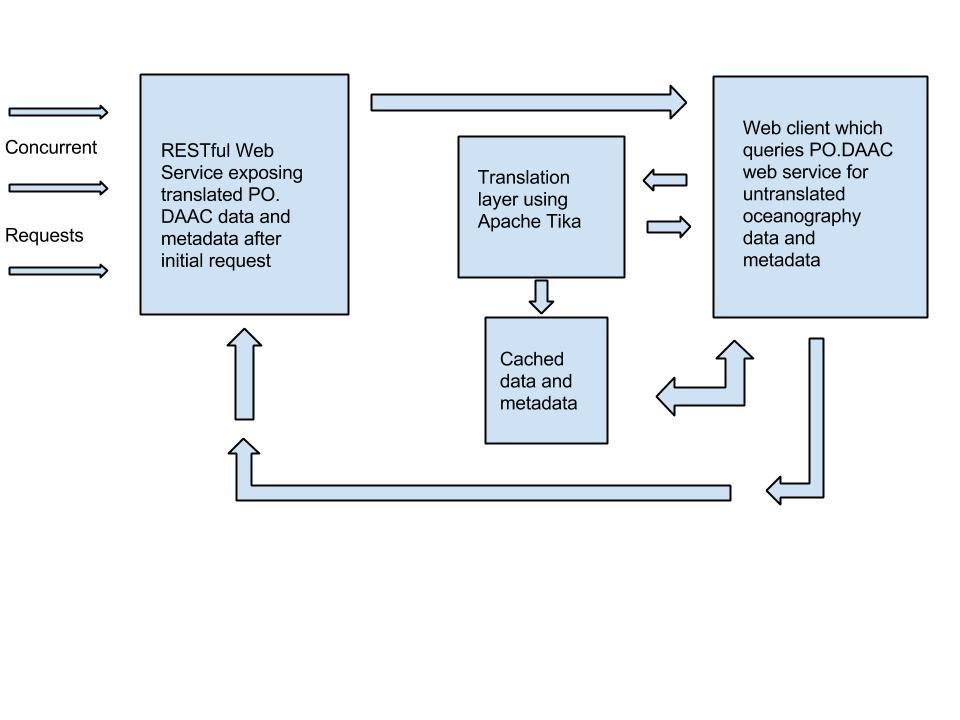
1.x.x - X new languages supported.

**Design**

The “flow” of this project is as such: Concurrent requests are made to a web service, which queries the existing PO.DAAC web service for untranslated data and metadata, and translates it on-demand for the client. It then exposes this data in whichever form requested (JSON or XML).

The client-facing service will be responsible for spawning off a new thread for each request, sending that request to the service-facing client. This request will correspond to a desire for multiple data sets, or some piece of a data set and its associated metadata. When the request on the other end is finished, this component will handle translating into a requested scientific data format (if necessary), and ensuring the information is set to the correct client.

The service-facing client will be responsible for extrapolating the desired data sets based on the request and asynchronously making requests to a distributed cache and/or PO.DAAC services. It will be responsible for translating non-cached data using Apache Tika and caching information if it is necessary (such as how frequently that data set is accessed). Finally, it will be responsible for filtering and merging translated information based on the request before sending the information back to the client-facing service.



**Tasks**

1. Deploy the Clojure demo to AWS or JPL environment
   1. Install whatever is necessary (such as Leiningen) onto the server environment.
   2. Deploy Clojure demo successfully.
   3. Test (likely ad-hoc) the demo to make sure it services requests correctly.
2. Create a component which exposes requested data to a client
   1. Create a ring/compojure layer which interprets client information in a RESTful manner.
      1. Create and run unit tests for this using ring-mock.
   2. Create a layer above the ring/compojure layer which spawns off a thread for a client, sending their request to the next layer.
      1. Create and run unit tests for this, idealing using ring-mock or another mocking library, to ensure basic functionality is met.
      2. Load test this to ensure the server can handle some large concurrent requests.
3. Create the component which keeps track of which client requested which data, and ensures that a thread spawned off for a request associates its data with the correct client who requested it.
   1. Create and run unit tests, ideally using ring-mock or another mocking library, to ensure basic functionality is met.
   2. Load test this to ensure the server can handle large sets of callbacks well.
4. Create a component which converts the requested data into the file format the user desires (HDF4, HDF5, XML, JSON, etc).
   1. Create and run unit tests which ensure data can be successfully converted into these formats.
5. Create an integration test, with an accompanying load test, to ensure that this end of the service can handle large amounts of clients concurrently.
6. Create a component which receives request information from the first component and extracts PO.DAAC data and metadata based on the request information.
   1. Create a layer which analyzes the request, determining what data and metadata the user requested and how it related to PO.DAAC services
      1. Create and run unit tests using sample request data for all PO.DAAC data set samples, and then with a handful of legitimate data set query values.
   2. Create a layer that, given valid request data, determines if information is already cached, or if PO.DAAC services will have to be queried with that data.
      1. Create and run unit tests using request information whose data is likely to be cached, information whose data is not likely to be cached, and use some kind of mocking mechanism to control data in a fake”cache” in the case that requested data is cached.
   3. Create a layer that can merge cached data with newly-translated data, supporting the case that some of the requested information is cached, and some is not.
      1. Create and run unit tests which verify that merging data is done correctly. Using mocking of some sort to merge cached and newly-generated translated data and metadata.
   4. Create an integration test which ensures each of the above parts run together properly.
7. Create a component that translates PO.DAAC data and metadata using Apache Tika.
   1. Create a layer which receives data from PO.DAAC, extracts metadata, translates it, and translates the actual data set.
      1. Create and run unit tests which test against the PO.DAAC sample sets and other generated information.
   2. Create a layer which caches translated data and metadata, if it is determined to be worthy of caching.
      1. Create an integration test which uses components from Task 2 to ensure the caching mechanism is working.
   3. Create a layer which sends data to part (c) of Task 2, allowing for translated data sets and metadata to be merged with cached data.
      1. Create and run an integration test which communicates with the Task 2 component.
8. Create an Integration test which ensures a user can query for data, have that request fulfilled by retrieving PO.DAAC data, translating it, and sending it back to the user in a requested format.
9. Create a System Load Test which makes many concurrent requests, and ensure that the system doesn’t buckle under the load.
10. Deploy releases to JPL’s environment on a frequent basis, with pieces of each of the above tasks included as needed.
11. For each deployed release, run Task 5 in their environment on a staging server.
12. For each deployed release, run a quick acceptance test with Lewis and/or other JPL staff to ensure requirements are met from their perspective.

**Risk Assessment**

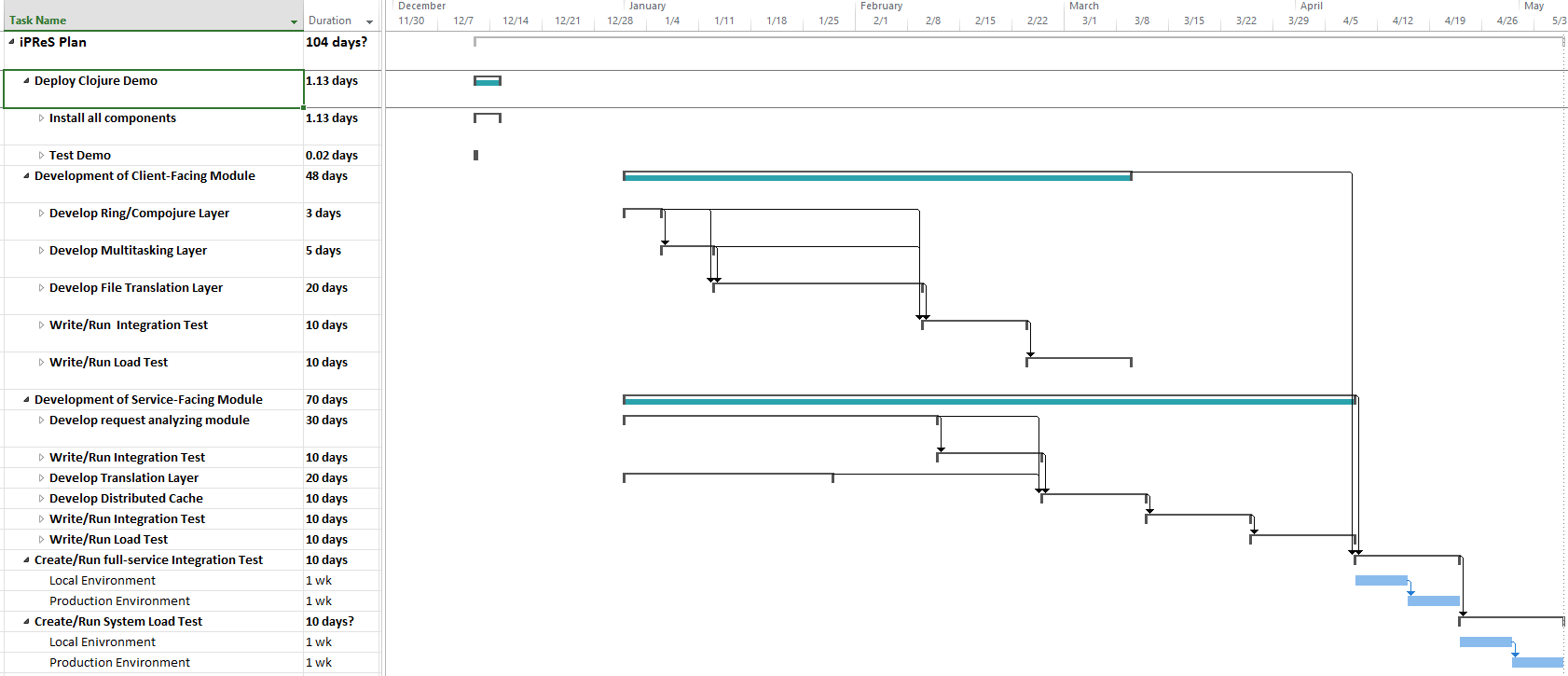
1. Failing to successfully set up a Clojure environment at JPL is a legitimate concern. Our plan for this is to migrate to using Java, because we know that a Java servlet can be run on their servers. We aim to figure this out as soon as possible by making a very basic version 1 release quickly during the development phase, so we aren’t impacted very much.
2. Not supporting all the desired languages is another legitimate concern. Our plan for this is to recognize where we stand relative to our plan, and adjust on a weekly basis. Before development, we will have a subset of the requested languages which represent a priority list. If we feel we cannot feasibly support every language, we will focus on the prioritized languages and build them with as high of quality as possible.

**Testing**

We plan on following a fairly modern practice by combining sensible pieces of Test-Driven Development and Continuous Integration. Our rules we follow are:

* A source control Pull is made each time a developer works on code
* All implementation code is written with corresponding test code (unit tests, integration tests, etc)
* All potential commits to the Github repository must pass all previously-passed tests before being pushed onto the current branch
* Integration testing of all modules are re-run when a new module is finished with development
* All Integration Tests must pass before running a System Test
* System Tests are run when all modules for a release are complete
* Acceptance Testing is done on a staging server when a release is being published
* Acceptance Testing is done on the production server after Acceptance Tests pass on the staging server

**Timetable**



Based on the Gantt chart created, the project proper is estimated to start January 1st, and end May 6th. This includes integration and load testing, which is done for each major task and some of their subtasks. As expected, the critical path is in the development of the service-facing part of the iPReS service, as it contains more components which need thorough testing. This leaves some slack time on the client-facing end to potentially work with some non-scientific data formats (CSV, PDF) if development does not go over the expected time frame.

**Team Roles**

All team members will have equal roles; there will not be a designated coder and tester, etc. We intend to be flexible, sharing workloads as equally as possible, and branching out as necessary to accommodate tasks as needed. That being said, due to the nature of this being a class, we will split up work.

Daniel will primarily be responsible for tasks in developing the client-facing module of the application. Phillip and Bhavik will primarily be responsible for developing the service-facing module, including the distributed cache. There will be a mostly collaborative work; however, we will collectively have ownership over all tasks and help each other out as necessary.

**Integration Plan**

Communication and making numerous, small updates on a frequent basis. The best way to coordinate work is to collaborate and make consistent, small, and numerous updates so that no sweeping changes are made by any one person.

**Signatures**

The signatures below confirm that this work is original and that the signees have produced, read and understood the above document in its entirety.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Lewis John McGibbney (Client Team)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Phillip Michael Carter (Engineering Team)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Bhavik Vikram Patel (Engineering Team)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Daniel Song (Engineering Team)