|  |
| --- |
| VIVO: Enabling National Networking of Scientists |
| VIVO Technical Specifications |
| How VIVO works on the inside |

|  |
| --- |
| The Members of VIVO: Enabling National Networking of Scientists  5/6/2010 |

Table of Contents

[I. Background 4](#_Toc262720281)

[II. Summary 4](#_Toc262720282)

[III. Data Storage 4](#_Toc262720283)

[A. Scalability 4](#_Toc262720284)

[i. RDF and OWL Representation 4](#_Toc262720285)

[ii. Reasoning 4](#_Toc262720286)

[B. File management 4](#_Toc262720287)

[C. Security 4](#_Toc262720288)

[i. Provenance 4](#_Toc262720289)

[IV. Semantic Capabilities 4](#_Toc262720290)

[A. Searching 4](#_Toc262720291)

[B. Linking External Taxonomies 4](#_Toc262720292)

[C. Inter-ontology Mapping 4](#_Toc262720293)

[D. Ontology Editor 4](#_Toc262720294)

[i. Ontology Change Management 5](#_Toc262720295)

[ii. Ontology Graph Management 5](#_Toc262720296)

[iii. Local Ontology Support 5](#_Toc262720297)

[V. National Network 5](#_Toc262720298)

[A. Aggregator Service 5](#_Toc262720299)

[B. National Network Search 5](#_Toc262720300)

[C. Linking External Individuals 5](#_Toc262720301)

[VI. User Interface 5](#_Toc262720302)

[A. Web Site Framework 5](#_Toc262720303)

[B. Administrator 5](#_Toc262720304)

[i. Menu Management 5](#_Toc262720305)

[ii. Manual Data Ingest 5](#_Toc262720306)

[iii. Templating 5](#_Toc262720307)

[iv. Display Configuration 5](#_Toc262720308)

[C. Individual User 5](#_Toc262720309)

[i. Data Review and Self Editing 5](#_Toc262720310)

[ii. Public/Private Data 5](#_Toc262720311)

[VI. Interface with External Systems 6](#_Toc262720312)

[A. Harvest Data 6](#_Toc262720313)

[i. Fetch Framework 7](#_Toc262720314)

[ii. Translation Framework 8](#_Toc262720315)

[iii. Scoring Framework 12](#_Toc262720316)

[iv. Qualification Framework 13](#_Toc262720317)

[v. Transfer Framework 13](#_Toc262720318)

[vi. IO Framework 14](#_Toc262720319)

[vii. Controller Framework 14](#_Toc262720320)

[viii. Data Sources 14](#_Toc262720321)

[VII. Linked Data and Open SPARQL Endpoints 17](#_Toc262720322)

[A. External SPARQL Endpoint 17](#_Toc262720323)

[B. Interface with Drupal 17](#_Toc262720324)

[C. Interface with Wordpress 17](#_Toc262720325)

[D. Interface with Sakai 18](#_Toc262720326)

[VIII. Authentication 18](#_Toc262720327)

[A. Shibboleth 18](#_Toc262720328)

[i. Administrator 18](#_Toc262720329)

[ii. End User 19](#_Toc262720330)

[B. Kerberos 19](#_Toc262720331)

[IX. Packaging 19](#_Toc262720332)

[A. Virtual Machines 19](#_Toc262720333)

[i. VIVO VM as for Testing/Example 19](#_Toc262720334)

[ii. VIVO VM as a Production Environment 19](#_Toc262720335)

[iii. VIVO VM as a Marketing Tool 19](#_Toc262720336)

[X. Risks 19](#_Toc262720337)

[XI. FAQ’s 19](#_Toc262720338)

# Version Control

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Author(s)** | **Change Description** |
| 1 | 5/27/2009 | Christopher Barnes, Christopher Haines, Yang Li, Narayan Raum,  Dale Scheppler, Nicholas Skaggs, Stephen V. Williams | Document Created |

# I. Background

# II. Summary

This functional specification will describe the VIVO system related to the end users installation and configuration. This specification is, at this point, neither feature complete nor comprehensive. As we encounter new user requirements, use case scenarios should be revised and/or added to the specification to guarantee that all situations are represented.

# III. Data Storage

## A. Scalability

### i. RDF and OWL Representation

### ii. Reasoning

## B. File management

## C. Security

### i. Provenance

# IV. Semantic Capabilities

## A. Searching

## B. Linking External Taxonomies

## C. Inter-ontology Mapping

## D. Ontology Editor

### i. Ontology Change Management

### ii. Ontology Graph Management

### iii. Local Ontology Support

# V. National Network

## A. Aggregator Service

## B. National Network Search

## C. Linking External Individuals

# VI. User Interface

## A. Web Site Framework

## B. Administrator

### **i. Menu Management**

### ii. Manual Data Ingest

### iii. Templating

### iv. Display Configuration

## C. Individual User

### i. Data Review and Self Editing

### ii. Public/Private Data

# VI. Interface with External Systems

## A. Harvest Data

What are the basic parts of the harvesting information from an external system? The system must connect to the external system and download the information. It must translate that data from its original schema to the VIVO ontologies schema. It must evaluate the information against the existing VIVO system looking for links and equivalent entities.  The system must also validate and sanitize the data allowing for only clean entries to pass into VIVO. Finally, the system must import the information into the VIVO Jena Triple Store.  These tasks make up the black boxes that are the process of the VIVO Harvester.



Figure : Overall Harvester Design

The harvester is broken into seven parts; Fetch, Translate, Score, Qualify, Transfer, I/O, and the Controller. The controller oversees the entire process. Through the use of configuration files, the harvester can be operated in many different modes. It can be configured to run through the entire process, or any combination of the individual parts.

* Fetch queries and ingests external data sources. This external data can come in many different forms – RDF, CML, RSS, databases, files, etc.
* Translate uses definitions to translate external data and output it in the desired format. The intended output format is VIVO ontology, but using a mapping file allows our translator to be open ended for converting to other ontologies. Ultimately a tool such as gloze will assist in the creation and translation of raw inputs to RDF ontologies.
* Score performs author disambiguation to match incoming data with people found in VIVO. The initial versions will implement defined rulesets, ultimately converging on a self-learning system.
* Qualify readies and cleans the data for insertion into VIVO. The rulesets will be externally stored and configurable.
* Transfer performs version mapping of the translated VIVO data, inserts entities into the VIVO data store and links entities to each other in VIVO.
* I/O handles connections between processes and abstracts all aspects of input and output; supporting writing to console, file or database.

### i. Fetch Framework

Develop the overall framework for performing fetches against external data sources for eventual harvesting to VIVO. Design centers around core functions, getRange(x,y), getNew(date), getModified(date), etc. Classes will exist for each protocol found.

* Architectural Design should support extension and reinvention
* Identify Typical Protocols (SOAP, restful, ODBC, RSS, etc)
* Identify Data protocols being transmitted (result sets, xml, objects, etc)
* Determine methods that will exist
* Identify methods for storing raw data (pre translate data) for evaluation

#### a. SOAP

SOAP stands for "Simple Object Access Protocol", and is a protocol specification for exchanging structured information in the implementation of web services. It relies on XML as its message format, and usually relies on other application layer protocols such as RPC and HTTP for message negotiation and transmission. The XML based protocol consists of three parts, an envelope, which defines what is in the message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing procedure calls and responses.

We will be providing an overall template for fetching from a soap interface. Methods will allow for extension by data source specific implementations when a generic soap interface will not work. Specific SOAP implementations will be named and classed by the resource they are accessing (pubMedFetch, for example.) SOAP implementations are partially dependent on the format and types of data available from each provider and may work in tandem with other fetch methods to obtain the required information.

#### b. ODBC

ODBC stands for "Open Database Connectivity", which provides a standard software API for using Database Management Systems. In layman's terms, this means that an ODBC connection can communicate with a database regardless of operating system, database type, or programming language. For the purposes of this document when we refer to ODBC data sources, we are referring to relational database systems, such as MySQL.

We will be fetching data from databases using the ODBC connection client, the specifics of these fetch methods will be system independent allowing the user to pass in the type of database just like you specify the type when creating the ODBC connection. This should allow for extension by data source specific implementations. We intend to make these data sources accessible through multiple methods, such as:

* Query Method
* Stored Procedure Method
* Table Select Method (specifying the tables to be pulled)

#### c. RSS

RSS, which is generally expanded to mean "Really Simple Syndication", is a family of web feed formats used to publish frequently updated works such as blog entries, news headlines, audio and video, in a standardized format. An RSS document (Which is generally referred to as a feed) includes full or summarized text, as well as metadata establishing details such as publication date and authorship.

We intend to access RSS feeds as an additional data source not only for articles written by VIVO users, but to assist with author disambiguation wherever possible. More detail will be added to this section as RSS feeds are identified.

#### c. Active Directory

Active Directory is the framework and structure used in Windows environments to store user information and handle access control. Active Directory profiles have both direct and indirect hints as to where someone works, what they do, what they have access to, and others. Active Directory data can also be used to generate eduPerson profiles, which can then be ingested into VIVO.

As more Active Directory data sources are identified, we will outline access methods here. Ideally there will be only one Active Directory fetch method, as it is a fairly unified front, however there is no telling whether or not one method will work for every Active Directory implementation.

### ii. Translation Framework

The translate framework takes in an input stream and an output stream. It converts the information found in the input stream from its predefined format into RDF in the format of the VIVO ontology. Included in the main VIVO harvester are methods for translating well known schema’s such as vCard, hCard, hResume, Medline, Bibtex, EduPerson, EduOrg, etc.



Figure - UML Diagram of Translation Framework

#### a. Translator

This is the main abstract class that defines basic functions of a translator object.

##### Inputs

**inStream** - *Input Stream*: The data to be translated is sent using an input stream. This stream can be cast into a FileInputStream or other input stream as defined by the extending class.

**outStream** - *Output Stream*: Following translation the transformed data the information is sent back to the controller through an output stream.

##### Methods

**Constructor:** Both a blank constructor, and a constructor that accepts an input stream and output stream are available depending on the users preference for setting object attributes.

**Execute**: This is just a stub for the models that extend Translator to fill in. It does test and return errors if the input stream and output stream are not set.

**Set In Stream:** Allows the programmer to reset the input stream of the object. This would allow the same object to transform multiple records, by simply changing the input stream and re running the execute method.

**Set Out Stream:**  This allows the programmer to reset the output stream of the object. This allows the program to have multiple output locations, and simple re-execute the transformation.

#### b. XSL Translator

This method extends the Translator Class. Many data sources return their data in the XML format. These sources use DTD files, Document Type Definition, to describe their schema to the outside world. For sources with a DTD, the VIVO harvester accepts a XSL file that defines the elements in the source data and their transformation to the VIVO ontology.

##### Inputs

**Translation File –** *File*: The translation file provides the mapping of the original schema into the VIVO ontology. XSL transformations use XPath language to navigate the xml document. XPath allows the document transverse forwards and backwards throughout the xmls nodes, and access data elements from children or predecessors. The XSL files are placed in the data maps folder of the Harvester and described in the configuration file so they can be passed to the XSL translation class. This allows the user to simply define a mapping instead of requiring them to write java code. Translation files exist for Medline, hCard, hResume, hGrant, NIH Grants.

##### Methods

**Constructor:** Similar to the Translator class’, constructors exist for both blank creation and passing all of the necessary properties.

**Execute:** The main method of our xsl translator, this method checks for the necessary properties and then calls the xml translate method.

**Set Translation File:** This method allows for the same object to use multiple translation files, depending on the xml schema that is returned.

**XML Translate:** The workhorse of the XSL translation object, XML translate is a private method that utilizes javax.xml.transform package in Java. The package simply requires to sources, the file to translate and the xsl fle. Using the TranformerFactory, the method sets the xsl file, then calls for the transformation passing the xml source and the output method.

#### b. Gloze Translator

Gloze is a Jena Contribution module that can be used to transform XML naturally into RDF. It maintains the order relationship of XML and properly maps entities and attributes to object and data properties based on whether their complex or simple types. Complex types map to object properties and simple types map to data properties. Directly mapping XML to RDF allows the vivo harvester to utilize RDF mapping tools and workflows to transform data that is either too complex to map using XSL or requires other data sources to map properly.

##### Methods

**Constructor:** Like the translator it extends this method takes in an input stream and an output stream or allows for blank construction.

**Execute:** the main method of the gloze translator, this method checks for the input and output streams.

**Gloze Translate:** Utilizing the methods and classes found in gloze this method passes in the input stream to the gloze module.

#### c. D2RQ

D2R allows us to map relational data into RDF. At this time the capabilities of this method are unknown. D2RQ allows systems to represent relational databases as RDF. This allows the VIVO harvester to create a sparql construct that queries the relational database and returns RDF elements that can be passed on to the VIVO system. This method requires more research into D2RQ before the final design is solidified.

#### d. RDF Translation Map

In RDF we can define a workflow using strictly Sparql that takes RDF data from one model and transforms it into a new ontology and model. This method is similar to SQL Stored Procedures that allow you to programmatically merge using only the SQL query language.

#### e. vCard Translation

vCard files are in a special format that requires the harvester to parse each line to determine the attribute. To enable similar data sources where the information is unable to be processed by XSLT, Gloze, or D2RQ, the translation framework includes a structures package that models basic VIVO classes and provides a method of setting each data or object property and then requesting RDF in the VIVO ontology’s schema.

##### Methods

**Constructor:**

**Execute:** Unlike previous methods this method checks for the inputs, runs the parse method, and then returns to the output stream the RDF output provided by the people class of the structures package

**Parse vCard:** This method builds a java object using the classes found in the Structures package based on the attributes found in the vCard. When complete this method returns to the execute method the Person object that was created.

#### f. Structures

##### Classes

**People:** The people class has basic properties about a person, and arrays of objects for information like Addresses, Phone Numbers, Email. Which a person may have multiple of.

**Addresses:**

**Phone:**

**Email:**

**Jobs:**

### iii. Scoring Framework

#### a. Scoring.java

Method used to score data. Data is assumed to be in a VIVO-like ontology and stored in a Jena model. Method can call any combination of scoring functions. Scoring function will attempt to match data to authors in VIVO. Several algorithms and threshold values can be utilized to determine when the match will be inserted into VIVO. In addition, data produced by this method can be stored in a separate scoring model.

##### Flow:

* + 1. Executed by controller
    2. Read in config, models and parameters
    3. Call each scoring algorithm listed in config
    4. Return Model containing scored statements

##### Inputs:

3 Jena Models

* 1 Model containing VIVO
* 1 Model containing Scoring data
* 1 Model containing things to be scored in VIVO-like ontology

scoreConfig.xml

* + - * Defines which functions (algorithms) to run against input data
      * Also defines attributes need for algorithms

##### Outputs:

**1 Jena Model:** 1 Model containing matched things to be inserted into VIVO

##### Functions:

100% Match

* Give citation a score of 100% base upon given SPARQL query. For example, email, or other unique identifier.
* uniqueMatch(Model vivo, Model scoringInput, Query attributeQuery)
* return Model

Machine Learning

* + - * Complete implementation of neural machine processed rules. Requires the creation of a training set, training data, and a method of obtaining rules.
      * neuralMatch(Model vivo, Model scoring Input, Ruleset rules)
      * return Model

PairWise Scoring

* Utilizes input parameters of weights and attributes to assign an overall score to each citation. Input parameters will be hand created and must be customized for each dataset.
* pairwiseScore(Model vivo, Model scoring Input, Ruleset rules)
* return Model

Natural Language Processing

* + - * Perform analysis of citation language to find similar citations in VIVO. Requires threshold value to place into vivo model.
      * evaluateLanguage(Model vivo, Model scoringInput, Ruleset threshold)
      * return Model

### iv. Qualification Framework

Data quality control occurs in the qualification framework. This framework will provide a system for administrators to provide data integrity checks for each individual harvester.

### v. Transfer Framework

The transfer framework will receive rdf data ready for inclusion in the local VIVO data store and insure version control by migrating elements to the current version and creating the necessary links to objects in the local VIVO datastore.

### vi. IO Framework

#### a. Data Repository Connection Subsystem

The system for connecting to data repositories will support establishing connections with Jena Models and Relational Databases as well as creating file readers and writers for xml, csv, and flat files. By encapsulating the functionality for connecting with each of these data repositories, the burden of each of these tasks is removed from the other components.

#### b. Networking Subsystem

Each component of the harvest process can be run on a separate computer to allow for load balance and increased resource availability. To allow this to happen, there needed to be a subsystem that would allow the controller to send commands to each component and for data to be sent between each computer.

The networking system on operates over a potential three different connections: one for sending Operations and Responses called the Command connection, the other for streaming data to the subsystem called the Input connection, the last for streaming data from the subsystem called the Output connection. The Input and Output connections between the controller machine and subsystem machine are only established if the subsystem needs a streaming data connection to and from the controller respectively. In such a case, before the subsystem executes its operation the controller sends either an InputConnect, OutputConnect, or DualDataConnect Operation to the subsystem. The subsystem responds with by preparing for the incoming connection(s) and sending a InputWait, OutputWait, or DualDataWait Response. Once the data connections have been established, if necessary, the controller machine sends an Operation corresponding to the task to be executed. The subsystem then executes the specified operation, reading data off the Input connection and streaming data back over the Output connections to the controller machine as necessary. If there is an error, the subsystem sends one of the many error Response types. Once completed, the subsystem sends a Success Response on the command connection. If one of the data connections was established, the controller machine sends a InputDisconnect, OutputDisconnect, or DualDataDisconnect Operation. The subsystem then disconnects the data connection(s) and sends a InputClosed, OutputClosed, or DualDataClosed Response.

### vii. Controller Framework

The Controller provides a centralized system for controlling each of the phases of the Harvest process. The Controller’s primary jobs are the initiation of scheduled tasks for the components of the Harvest system and correctly routing data from each component to its destination, be it another component, a data repository, or combinations of the two.

The Controller is configured using an XML file which contains information about each task for each component. Each task is detailed in the configuration file: scheduled time, server to execute the task, where input data comes from, where output data goes, and any additional parameters the subsystem may require to execute the task.

### viii. Data Sources

Linked to the main framework is the implementation of harvesters for specific data sources. These data sources will be the focus during development of the various Framework components. Certain components of the framework may see a delay in construction based on when the dependant data source is scheduled for implementation.

#### a. NIH Articles: PubMed & Entrez Utilities

##### Estimations & Capacity Volumes

Our harvester system is presently estimated to ingest 19,000,000+ records at an average of 8 KB per citation, which we estimate will require 190 GB of disk space. We have researched PubMed and found the set standard for PubMed is 3 requests per second. The automated schedule for PubMed is from 9PM-5AM, at any time before 9PM and after 5AM queries are only allowed to pull 100 records per request.

##### PubMed Citation Locker/Endpoint

The first step in creating a PubMed citation endpoint is to retrieve citations out of PubMed.  Previous methods have only retrieved citations on a searching basis using author name. For this system, we will be utilizing the NCBI Eutils to request all citations from PubMed.  Several prototype systems will be initially created to test various parts of the process.

The first prototype will test the basic functions of requesting citations from PubMed.  It will explore the rate at which we can request citations from PubMed, the response time from PubMed for each request, and the average size of the files.  The prototype will also explore utilizing a configuration file that specifies information such as last download date and whether storing or retrieving the min and max PMID is more efficient.

A second prototype system will explore updating citations from PubMed.  It will need to answer questions such as “How do we determine which records have been updated?” or “Does our method for storing citations affect our ability to determine if a citation has been updated?”

Lastly we’ll need to convert a basic Medline XML citation into RDF.   There are various methods for converting XML to RDF such as XSLT files and the Gloze library. We’ll need to try these methods and determine which method leaves us with correct data, which method is the easiest to retrieve data from, and which method is the most efficient for the system. We may also test distribution of this data in an attempt to provide parallelization for author disambiguation across the VIVO network of institutions.

##### Author Scoring

The purpose of the Author Disambiguation component is to produce a score as to the probability of each author in VIVO being the author of each article from PubMed..

The first prototype will use a simple rule set to apply a score to the linkage of each author with each publication. It will use information about the institution and about the author in VIVO. A second prototype will expand this approach, combining the known information with the expanding list of publications it’s already matched with as part of the data about each author. In addition, administrators should be able to set threshold values to define which matches will be created in VIVO.

Another prototype must test the ability for administrators to tailor the program. Basic information such as school name, acronym, and email domain will be stored in a configuration file. This file will indicate how each piece of information is scored by the disambiguation program. A prototype must be able to read in this file and perform its disambiguation in the manner indicated by the configuration file.

Lastly, we will attempt to create a neural network that takes in the user’s settings and changes the order that it scores linkages as certain patterns prove more useful. This piece is crucially dependant on user feedback from inside vivo. Once this data is collected and made available, the implementation becomes possible. This is not a trivial task, and requires enhancements to the UI interface. While prototypes will be built and tested, this method may not be mature enough by the time the tool is first released and may require feedback from the VIVO community before it can be completed.

##### VIVO Harvest Interface

 Both an intermediary step and a final step in the PubMed harvest process is interfacing with VIVO.  The process needs to access Authors in VIVO to determine if a given author in PubMed is in a VIVO system. And it needs to import and update those citations that meet the minimum criteria for importation. Our prototypes must explorer two areas; querying VIVO and modifying VIVO.

 To better tailor the overall process, we must determine the best method for querying authors from VIVO.  To increase response time it may be better to store a copy of the specific information we need from VIVO about an author in a separate model away from the main VIVO installation. However, this comes with the overhead of maintaining a separate copy and how to best refresh it.  Time spent refreshing the copy model may offset the increased time it takes to query the main model, making it more efficient to just query VIVO directly.  A query prototype will explore all of these options and give us the necessary data to determine the architecture of our program

Finding the right match and inserting the information into VIVO sounds simple, and prototyping it should be.  However, this is a process we will repeat as we create more harvest programs.  The data from those programs will be different. So we should create a process that uses configuration files to take in data of one structure and place it into VIVO.  We will also need to prototype methods of update.  If a citation is updated in PubMed, and it already exists in VIVO, we will need to perform an update.  Do we check field by field or can we use a SPARQL update? This question will be answered as we prototype the update system.

#### b. NIH Grants: RePORTER

#### c. NSF Articles: Researcher.gov

#### d. NSF Grants: Award Search and/or Fastlane

#### e. ISI Publication

#### f. Scopus Publications

#### g. PeopleSoft

#### h. DSR

#### i. IRB

#### j. Sakai

#### k. BlackBoard

#### l. OAI Repositories

# VII. Linked Data and Open SPARQL Endpoints

## A. External SPARQL Endpoint

Provide ready access to VIVO data for external applications

* Data visibility control through graph management
* Address security and scalability implications of open access
* Explore using Joseki or Sesame
* Create tutorials for users install and use either Joseki and/or Sesame
* Research Data Security methods for non-visible entities and/or properties

## B. Interface with Drupal

Provide examples of interfacing with VIVO data in a Drupal installation. This is intended for institutions to disseminate VIVO data to their departments for use on departmental pages.

## C. Interface with Wordpress

Create an example of retrieving RDF data and loading it into a Wordpress site. This demonstration and instruction would be used by institutions/departments whose sites are completely done in wordpress. It may also serve as an example for non-wordpress sites, depending on the need for wordpress plugins.

## D. Interface with Sakai

# VIII. Authentication

Establish a framework for creating authentication modules for VIVO. The two main sources of authentication for VIVO are Kerberos and Shibboleth. This framework should allow for the creation of additional authentication modules as new institutions with different methods of authentication implement VIVO.

## A. Shibboleth

There are two types users in the current VIVO system:

1. Administrator
   1. These kind of users are pre-defined in VIVO “user” table.
   2. The users have different privileges based on the groups they are assigned to.
2. End User
   1. These kind of users are regular web users, such as, faculty, researchers, and staff, etc.
   2. These kind of users should NOT be defined in the VIVO user table.

The current VIVO system’s login can only validate Administrator. The current VIVO source code does NOT support login for End User.

### i. Administrator

I don’t think it is necessary to add Shibboleth for Administrator. But if we do, the procedure is simple based the VIVO released source code:

1. Setup Shibboleth client on VIVO web server (already done by Narayan)
2. If a user tries to access login page, redirect the user to Shibboleth authentication page.
3. After successful login on Shibboleth, the user is redirect back to VIVO login process.
4. Get the header information returned from Shibboleth and pass the unique user id (such as glid) to the customized Authentication class.
5. If the unique user id (such as glid) was defined in the user table, the user will be authenticated by the customized Authentication class and redirect to the home (or profile) page.

### ii. End User

The Shibboleth really should target to End User. Since we do NOT have pertinent source code from Cornell, we will assume the following procedure:

1. Setup Shibboleth client on VIVO web server (already done by Narayan)
2. If a user tries to access login page, redirect the user to Shibboleth authentication page.
3. After successful login on Shibboleth, the user is redirect back to VIVO login process.
4. Get the header information returned from Shibboleth and pass the unique user id (such as glid) to the customized Authentication class.
5. The user will be “simply” authenticated by the customized Authentication class and redirect to the home (or profile) page.

## B. Kerberos

Collaboration with Cornell as they use Kerberos.

# IX. Packaging

## A. Virtual Machines

### i. VIVO VM as for Testing/Example

### ii. VIVO VM as a Production Environment

Create a Virtual Machine for a production environment and document the process of taking the blank VIVO installation and making it production ready.

### iii. VIVO VM as a Marketing Tool

Provide a Virtual Machine and method of installation for the marketing team to use in promoting VIVO. Specifications will be drawn up by the marketing team in implemented by the packaging team.

# X. Risks

# XI. FAQ’s