OpenPolarServer (OPS) - An Open Source Spatial Data Infrastructure for the Cryosphere Community

By

Kyle W. Purdon

Submitted to the graduate degree program in Geography and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

____

Chairperson Dr. Xingong Li

____

Dr. Terry Slocum

____

Dr. David Braaten

Date Defended: April 17, 2014

UMI Number: 1559520

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI 1559520

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

The Thesis Committee for Kyle W. Purdon –

certifies that this is the approved version of the following thesis:

OpenPolarServer (OPS) - An Open Source Spatial Data Infrastructure for the Cryosphere Community

____

Chairperson Dr. Xingong Li

Date approved: April 17, 2014

Abstract

The Center for Remote Sensing of Ice Sheets (CReSIS) at The University of Kansas has collected approximately 700 TB of radar depth sounding data over the Arctic and Antarctic ice sheets since 1993 in an effort to map the thickness of the ice sheets and ultimately understand the impacts of climate change and sea level rise. In addition to data collection, the storage, management, and public distribution of the dataset are also one of the primary roles of CReSIS. The OpenPolarServer (OPS) project developed a free and open source spatial data infrastructure (SDI) to store, manage, analyze, and distribute the data collected by CReSIS in an effort to replace its current data storage and distribution approach. The OPS SDI includes a spatial database management system (DBMS), map and web server, JavaScript geoportal, and application programming interface (API) for the inclusion of data created by the cryosphere community. Open source software including GeoServer, PostgreSQL, PostGIS, OpenLayers, ExtJS, GeoEXT and others are used to build a system that modernizes the CReSIS SDI for the entire cryosphere community and creates a flexible platform for future development.

Acknowledgements

I acknowledge the use of data and/or data products from CReSIS generated with support from NSF grant ANT-0424589 and NASA grant NNX10AT68G.

Table of Contents

1 Introduction 1

2 Data 2

2.1 CReSIS Data 2

2.2 Community Data 4

2.3 Reference Data 6

3 Pre-OPS Spatial Data Infrastructure 7

3.1 Data Storage Formats 7

3.2 Data Distribution Methods 10

3.3 Data Access Methods 11

4 Problems and Objectives 13

4.1 Data Distribution Issues 13

4.2 Data Storage Issues 14

4.3 Project Objectives 17

5 OpenPolarServer System Structure and Components 18

5.1 Spatial Data Infrastructures 18

5.2 OpenPolarServer System Structure 19

6 OpenPolarServer SDI Implementation 25

6.1 Free and Open Source Software (FOSS) 26

6.2 CentOS Linux 26

6.3 PostgreSQL and PostGIS 26

6.4 Apache HTTP and Apache Tomcat 30

6.5 Geoserver 32

6.6 Django 42

6.7 Clients 60

6.8 Vagrant and VirtualBox 83

7 Usability Analysis 87

8 Conclusions 89

9 References 91

10 Appendices 95

10.1 Source Code 95

10.2 Web Application Links 95

1 Introduction

"The Center for Remote Sensing of Ice Sheets (CReSIS) is a Science and Technology Center established by the National Science Foundation (NSF) in 2005, with the mission of developing new technologies and computer models to measure and predict the response of sea level change to the mass balance of ice sheets in Greenland and Antarctica. The NSF's Science and Technology Center (STC) program combines the efforts of scientists and engineers to respond to problems of global significance, supporting the intense, sustained, collaborative work that is required to achieve progress in these areas. CReSIS provides students and faculty with opportunities to pursue exciting research in a variety of disciplines; to collaborate with world- class scientists and engineers in the US and abroad; and to make meaningful contributions to the ongoing, urgent work of addressing the impact of climate change [1]." Since 1993 CReSIS, or precursor groups to CReSIS, have collected over 700 TB of data in the Arctic and Antarctic using a suite of radars developed at the center [2]. The distribution of this dataset is one of the core duties of CReSIS. As a distributor of such an important dataset to the cryosphere community, and to the broader field of climate research, simplifying the process of data retrieval for data users and streamlining the process of data creation for the scientists and staff at CReSIS are two important and unresolved goals. In this work I present a new system, OpenPolarServer (OPS), which presents a resolution to these goals.

The primary goal of this work is to develop a free and open source SDI capable of storing, managing, creating, analyzing, and distributing the dataset collected by CReSIS in a way that provides ideal conditions for data users and primary producers of polar remote sensing data. In addition to the primary goal, a secondary goal is to design the proposed OPS in

a way that allows the entire cryosphere community to provide new datasets for inclusion in the system. To achieve the secondary goal special attention was paid to generalize all of the OPS components. In the following sections I will introduce the current SDI at CReSIS and two projects that predate this work. I will then review the literature of SDI, geoportals, cryosphere data, and free and open source software. Finally I will present the OPS in its entirety.

2 Data

2.1 CReSIS Data

CReSIS is the primary data contributor to the OPS [3]. Since 1993 CReSIS, or precursor groups to CReSIS, have collected over 700 TB of data in the Arctic and Antarctic using a suite of radars developed by the center [2]. Most of this data is in the form of radar echograms showing cross-sections of the Arctic and Antarctic ice sheets. These radar echograms are designated as Level 1B (L1B) data defined by NASA as “data that have been processed to sensor units [4].” Fig. 1 shows a radar echogram with major features labeled and a map indicating the location of the radar echogram. The radar echogram offers a raw data product from which usable information must be extracted. For CReSIS the ice surface layer and ice bottom layer are the target information for extraction. The extraction process is carried out using the Data Picker, a custom MATLAB software developed at CReSIS. It is a tool which allows for manual digitization (with some automation) of the ice surface and ice bottom layers resulting in a point based dataset with the primary attribute of layer elevation. The digitization process results in a

dataset of points, each with a set of attributes that will be discussed later. These points are designated as Level 2 (L2) data defined by NASA as “derived geophysical variables at the same resolution and location as Level 1 source data [4].” Fig. 2 shows the radar echogram in

Fig. 1 with the digitized ice surface and bottom layers and an inset that shows the digitized point objects. This digitized ice surface and ice bottom is the primary data product stored in the OPS database.

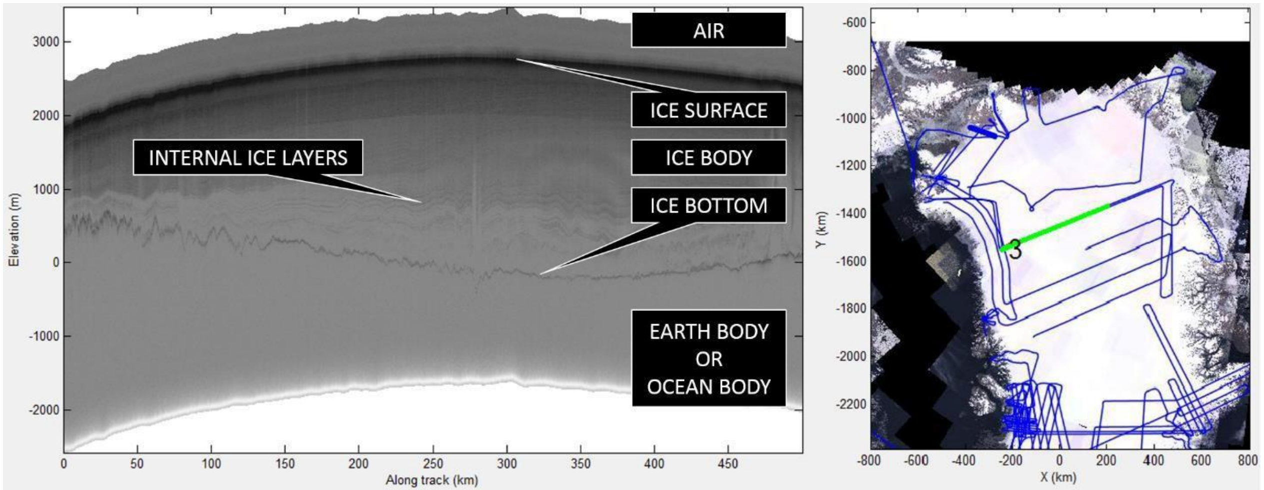

Fig. 1: A CReSIS L1B echogram (left) with its location (green line) identified via a map of Greenland (right). Major features are labeled on the radar echogram.

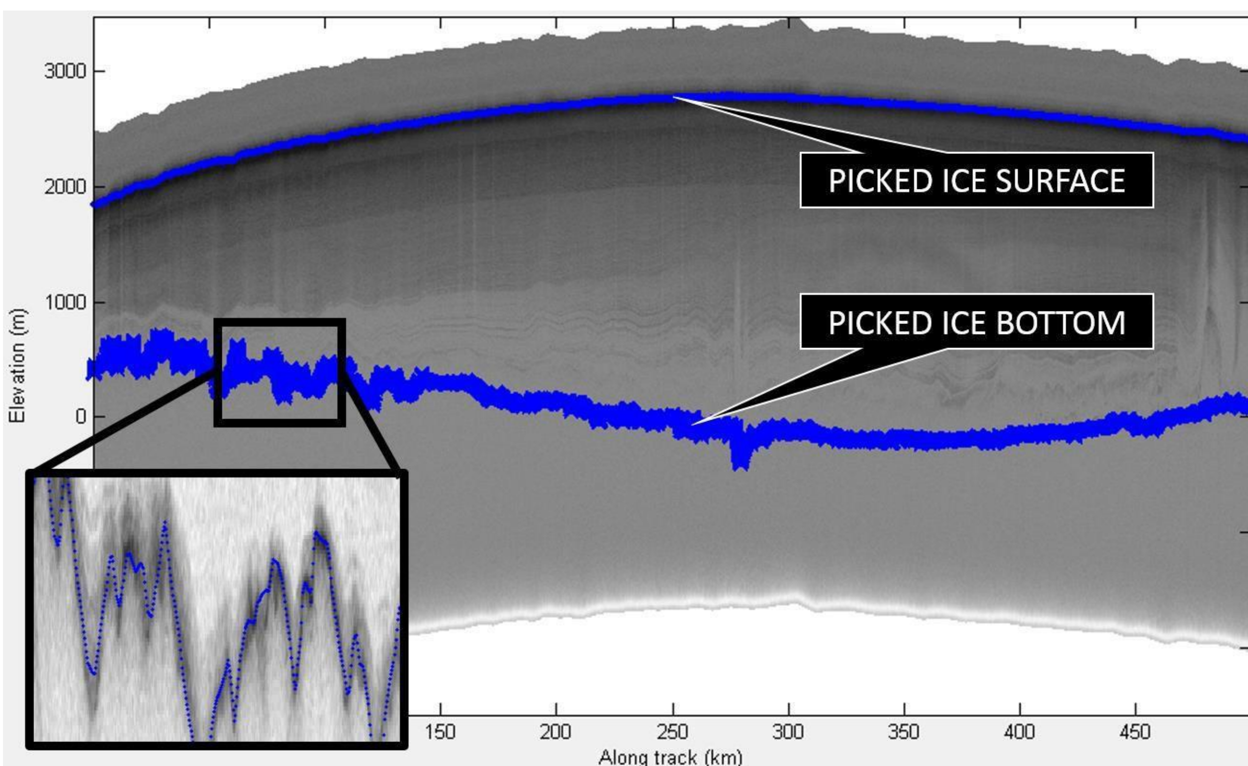

Fig. 2: A CReSIS L1B radar echogram (Fig. 1) with digitized (picker) layers (blue)

2.2 Community Data

In addition to the primary data product, described in section 2.1, there are additional layers that can be extracted from the radar echograms by manual and automated digitization methods. There is a community of cryosphere scientists that are developing methods for digitizing the internal ice layers present in the CReSIS L1B dataset. As stated in the introduction it is a primary goal of the OPS to support not only the CReSIS dataset but also a broader selection of polar remote sensing data. The digitized point data representing internal ice layers is one of these non-CReSIS generated datasets. A current venture is underway at The University of Texas to automatically extract the entire radio stratigraphy (internal ice

layers) of the complete CReSIS L1B dataset [5]. Fig. 3 shows preliminary results of the automated radio stratigraphy derived from an L1B radar echogram from the 2011 Greenland P3 CReSIS dataset.

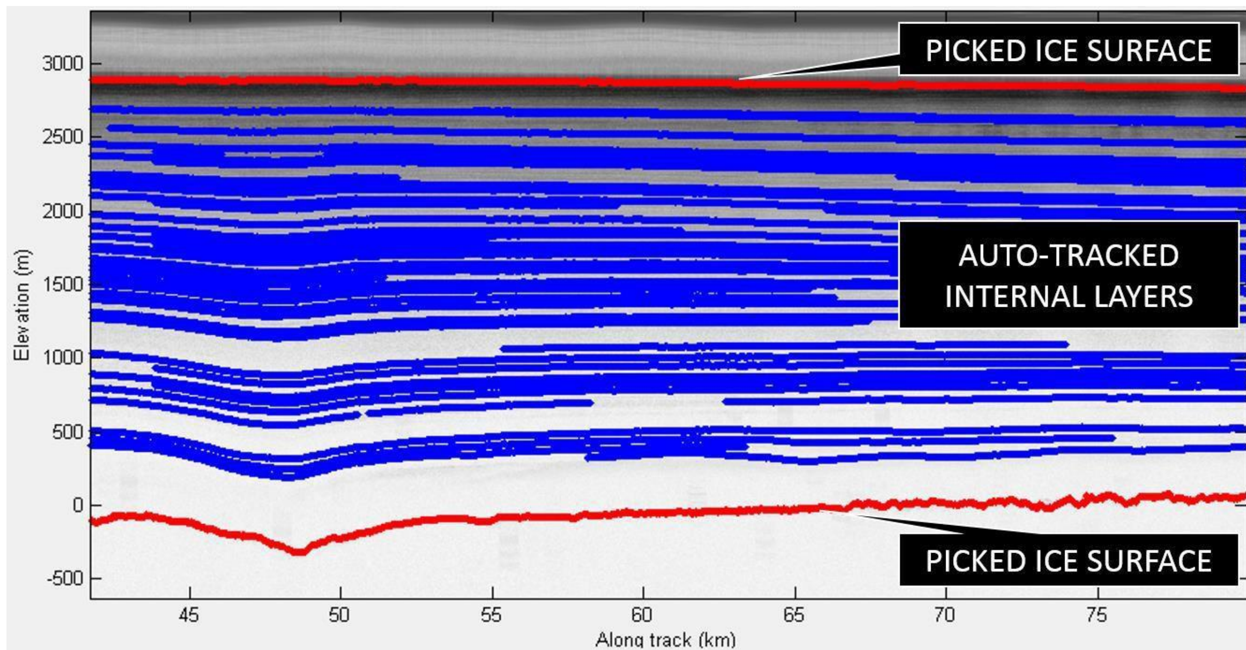

Fig. 3: Automatically digitized internal ice layers (blue) and CReSIS digitized ice surface and ice bottom (red).

In addition to non-CReSIS data generated from the L1B radar echograms, another source of data for the OPS is two LiDAR instruments, the NASA Airborne Topographic Mapper (ATM) [6] and Land, Vegetation, and Ice Sensor (LVIS) [7], which collect data on the same aircraft as CReSIS radars during various field data collection campaigns. These lidar sensors provide a high precision measurement of ice surface elevation that can be stored in the OPS database and displayed on CReSIS radar echograms. Fig. 4 shows a CReSIS radar echogram with a manually digitized ice surface layer and the ice surface measurement from the

ATM lidar as a secondary layer. These examples of non-CReSIS layers are just a sample of the potential community inputs to the OPS. There are other research groups collecting related datasets that could benefit from the features the OPS SDI provides. It is the author's hope that with the OPS implementation described in this thesis the community can finally come together and grow within a single framework.

2.3 Reference Data

In addition to the datasets discussed in sections 2.1, and 2.2, the OPS is reliant on a collection of reference data, primarily in the form of raster imagery and vector geographic boundaries. Fig. 5 shows a few examples of the datasets that serve as reference data for the OPS. In general the reference datasets include satellite imagery [8][9], raster velocity data [10], bedrock digital elevation models [11][12], and vector coastline data [13]. A complete listing of reference datasets is included in the OPS software.

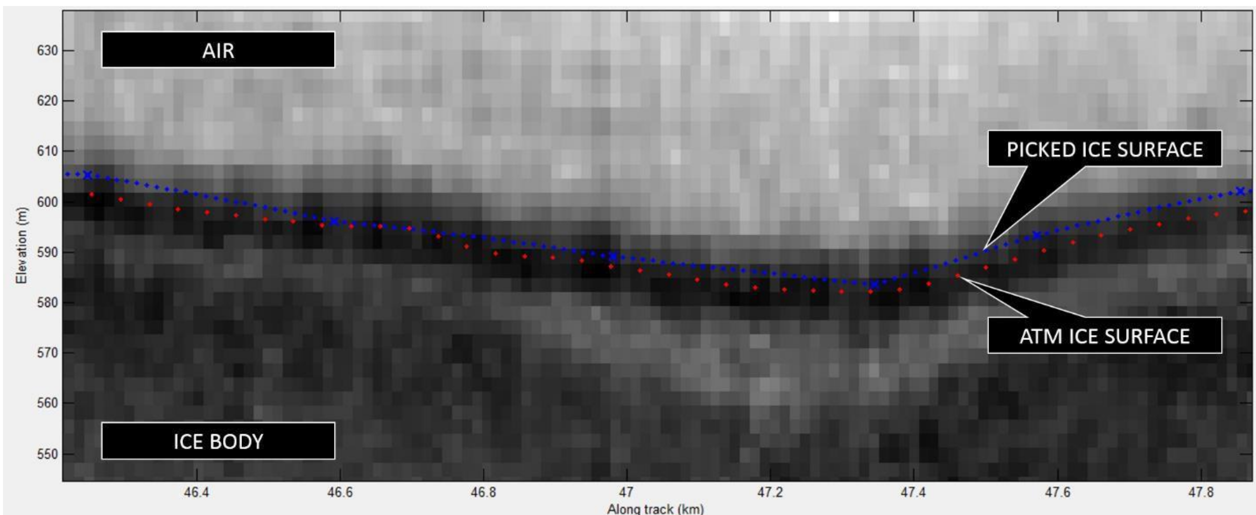

Fig. 4: CReSIS digitized ice surface (blue) and ATM collected LiDAR ice surface data (red).

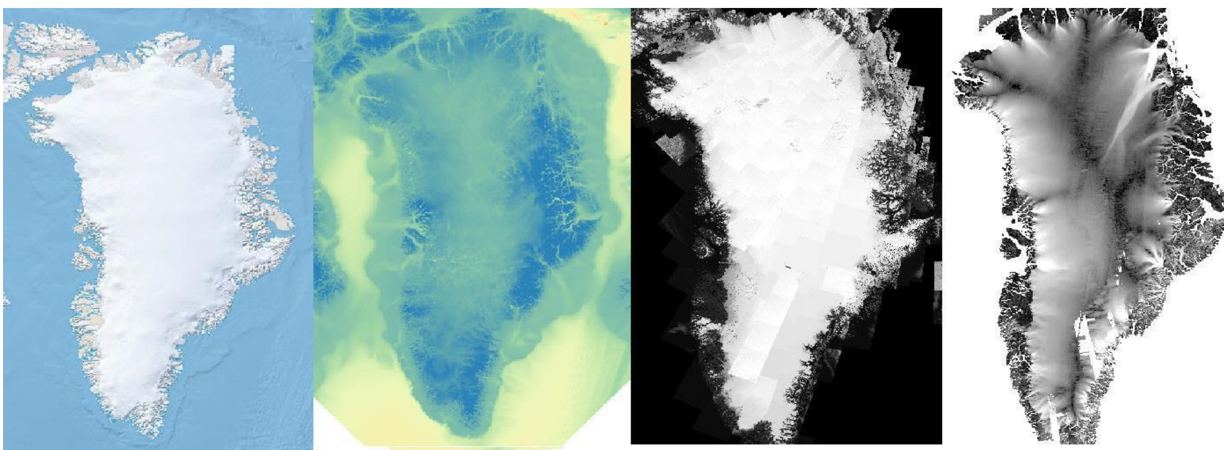

Fig. 5: A sample of reference data included with the OPS. Shown from left to right is Natural Earth Imagery, Bamber V3 Bedrock Elevation, Landsat-7 Imagery, and Joughin Ice Surface Velocity.

3 Pre-OPS Spatial Data Infrastructure

This chapter provides a review of data distribution at CReSIS prior to the integration of the OPS. I first introduce the data storage formats for each of the primary CReSIS datasets (L1B radio echograms, L2 ice surface and bottom layers), as well as the formats for the CReSIS derived internal ice layers. Then, I present the chosen methods of distribution for the primary CReSIS datasets and finally discuss data access methods, including two projects that predate the creation of the OPS.

3.1 Data Storage Formats

The primary data storage format for CReSIS L1B and L2 datasets is the MATLAB binary file format (.mat) commonly referred to as a MAT file [14]. The MAT file format is a

proprietary binary file format used by the high-level scripting language and interactive environment for numerical computation, MATLAB. CReSIS chose this format as its primary data storage container because all of the data processing, digitization (picking), and analysis is performed using MATLAB. CReSIS designates two primary types of MAT files called data files and layerData files. The data files store the L1B radar echogram values and the layerData files store the L2 ice surface and bottom values. CReSIS data is divided by day and then each day is divided into segments. Each segment is further divided into frames. There is a single layerData file per frame. There is a minimum of one data file per frame but there can be more if additional radar echogram processing types are included. In addition to the two primary files (data, and layerData) CReSIS also stores additional information about the data collection path geometries in the MAT file format. The data collection paths or tracks are referred to as flightlines and are stored in MAT files called gps, records, and frames files. There is a single GPS file per day, a single records file and a single frames file per segment. GPS files store time and position data from the onboard GPS instruments. Records files store processed time and position data. Frames files store the integer indices of data in records files marking the subdivision of segments into frames. Fig. 6 shows a graphical representation of the gps, records, frames, data, and layerData files for a single segment of data.

While the MAT file format is convenient for users of MATLAB there are many other common data formats that CReSIS must provide to end users including (but not limited too) comma-separated values (CSV), keyhole markup language (KML), and joint photographic experts group (JPEG). To create these formats CReSIS has created a collection of tools called the CReSIS toolbox. This toolbox contains many functions and scripts written in MATLAB that convert from the MAT file format to outputs such as CSV, KML, JPEG and more. Once

this collection of output files is created it is made available to users through two primary services: the CReSIS FTP [15] and the National Snow and Ice Data Center (NSIDC) data archive [16].

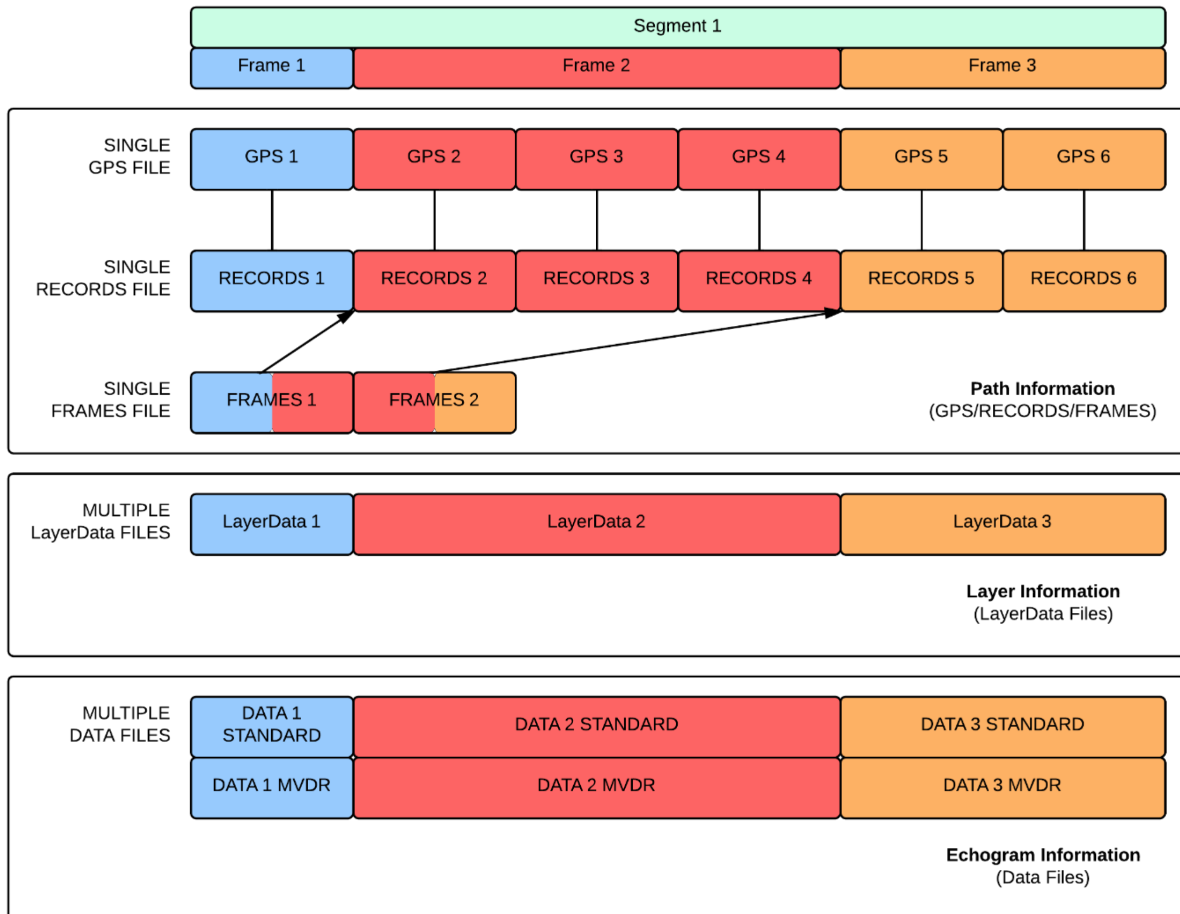

Fig. 6: An example set of CReSIS data for a single data collection segment. 3 files (1 GPS, 1 RECORDS, 1 FRAMES) store the flight path data, 3 files (1 layerData per frame) store the digitized layers, and a minimum of 3 (1 per frame per processing type) store radar echograms.

3.2 Data Distribution Methods

File Transfer Protocol (FTP) is one of the standard methods for transferring files over the internet from one computer to another. CReSIS implements a standard FTP server which exposes an organized directory of files to the web. This allows users to connect to and transfer any files from the FTP server to their own computer. This is a simple and fast method of data transfer. The CReSIS FTP was the primary method of data distribution prior to the integration of the OPS. All of the files discussed in the previous section (gps, records, frames, data, and layerData) as well as the output file formats (CSV, KML, JPEG …) are served through the CReSIS FTP. Fig. 7 shows a standard season directory on the CReSIS FTP. The NSIDC data archive is a hub for the archival of all cryosphere data. In the background, the NSIDC website is essentially just an FTP. However NSIDC implements a data portal with a more organized approach of navigating and selecting datasets for download. The NSIDC Operation Ice Bridge (OIB) data portal is discussed in section 3.3.

Filename	Filetype	Last modified	Permissions	Owner/Group
..				
CSARP_layerData	File folder	1/23/2013	drwxrwsr-x	2414 2414
CSARP_mvdr	File folder	1/23/2013	drwxrwsr-x	2414 2414
CSARP_qlook	File folder	1/23/2013	drwxrwsr-x	2414 2414
CSARP_standard	File folder	1/23/2013	drwxrwsr-x	2414 2414
csv	File folder	1/23/2013	drwxrwsr-x	2414 2414
csv_good	File folder	1/23/2013	drwxrwsr-x	2414 2414
images	File folder	1/23/2013	drwxrwsr-x	2414 2414
kml	File folder	1/23/2013	drwxrwsr-x	2414 2414
kml_good	File folder	1/23/2013	drwxrwsr-x	2414 2414
pdf	File folder	1/23/2013	drwxrwsr-x	2414 2414

Fig. 7: A standard season directory on the CReSIS FTP. The directories shown contain all of the various file types (MAT, CSV, KML ...) and are further organized in subdirectories not shown in this image.

3.3 Data Access Methods

In addition to simply browsing the CReSIS or NSIC FTP for data there are additional tools that allow users to download the data. Of particular interest are three data access methods: Geographic Search GUI, PolarGrid, and the NSIDC Operation Ice Bridge (OIB) Data Portal. These three tools represent the most prominent data access methods available to the cryosphere community for retrieving CReSIS data. Fig. 8 shows a preview of the three tools.

This NSIDC OIB Data Portal [17] exposes the NSIC FTP contents to a map interface and allows users to visualize what data is available on the NSIDC FTP. It offers some basic filtering capabilities such as the date of collection and the data source but does not offer data subsetting. The end product of this tool is still one or many files (in various formats) retrieved from an FTP.

The PolarGrid Cloud GIS [18] was developed as part of the broader PolarGrid project at Indiana University [19]. This service was intended to mature into a fully functional geoportal which has not yet occurred. As of the time of this thesis there is no active support or development on the PolarGrid Cloud GIS and it can be considered EOL (end of life). The PolarGrid Cloud GIS service offered similar functionality to the NSIDC OIB Data Portal but was designed to interact with the CReSIS FTP. The PolarGrid Cloud GIS project was handed over to CReSIS and though most of the work was scrapped the idea became the OPS.

Geographic Search and its successor Geographic Search GUI are MATLAB-based tools and offer an application programming interface (API) to the CReSIS FTP. Geographic Search offeres a simple scripting interface to the FTP allowing a user to manually enter a boundary of coordinates, a list of datasets, and an output format to retrieve a subset of data

from the FTP. The tools work by downloading all of the required data from the FTP and using MATLAB to subset and output the dataset. Geographic Search GUI [20] was developed as a enhancement to the Geographic Search tool. It added an interactive command line input, a larger range of filtering and output options, and a graphical user interface (GUI) to its predecessor.

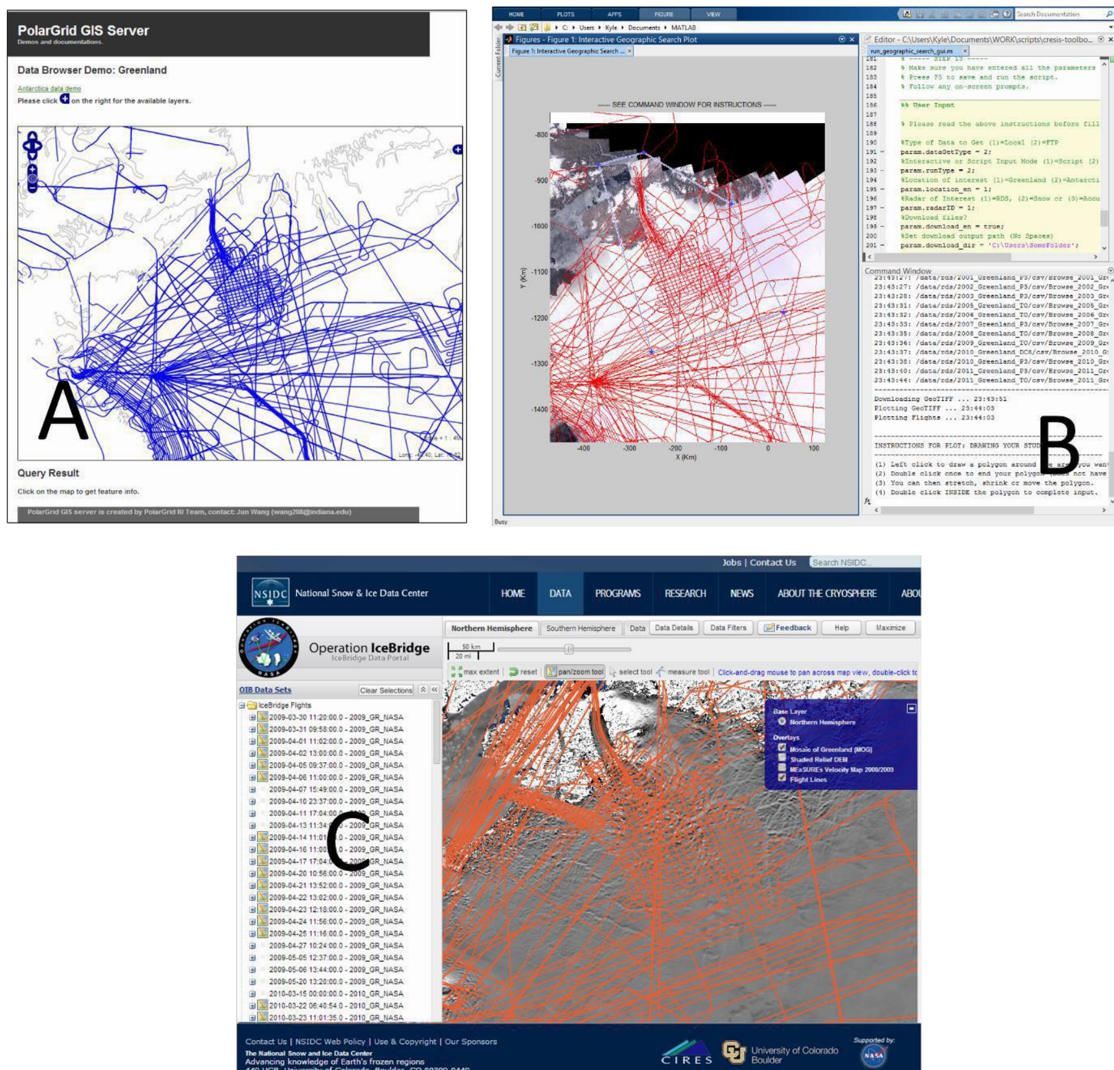

Fig. 8: Previews of various tools that give users access to CReSIS data. The PolarGrid Cloud GIS (A), MATLAB Geographic Search GUI (B), and the NSDIC OIB Data Portal (C) are shown.

4 Problems and Objectives

4.1 Data Distribution Issues

The primary method of data distribution for CReSIS is currently the CRESIS FTP website discussed in section 3.3. Let us imagine the following common data use task: *Using the CReSIS FTP site download all ice thickness data in some specified region.* With this task in mind the user might follow the

KML Search Method:

1. Download and load all of the posted KML files from the ftp site.
2. Use the KML metadata to determine the segment IDs for data in the region of interest.
3. Navigate the ftp site and download each file matching the noted IDs individually.
4. Merge, load, clip and finally use the downloaded

files. or the JPEG Search Method:

1. Navigate the ftp site and visually identify (using posted JPEG images) the data in the region of interest and note the segment IDs.
2. Navigate the ftp site and download each file matching the noted IDs individually.
3. Merge, load, clip and finally use the downloaded files.

A third method, while not recommended, is that a data user downloads an entire local copy of the FTP site and uses their own custom tools for data extraction and processing.

To strengthen the point that none of the above methods cater to usability, an example using more specific selection of data for download follows. If a user were to follow the KML Search Method to download CReSIS data from 1993-2013 in the Petermann glacier region they would first need to download the approximately 170 GB of posted KML files, then find approximately 360 unique segment IDs and would finally have to navigate to approximately 10

different directories on the ftp to download the 360 unique files in one of the provided formats (MAT, CSV, …). Following this download they would need to merge all of the downloaded data and subset the resulting dataset to their specific study boundary.

The MATLAB Geographic Search GUI alleviated some of the issues faced by a FTP user. Geographic Search GUI presented as alternative solution to the use of the basic CReSIS FTP site. Unfortunately, there is no recorded user data to confirm or deny that Geographic Search GUI is being used; however the fact that CReSIS FTP usage has not declined significantly shows that the problems are not fully addressed by the tool [2]. Geographic Search GUI is hindered by its implementation as a MATLAB tool; however it is the only tool prior to the integration of the OPS that allowed for spatial subsetting of the output dataset.

4.2 Data Storage Issues

The primary method of data storage for CReSIS is the MATLAB binary files discussed in section 3.1. File-based storage has many disadvantages, some of which are:

1. Additional libraries/code are required to load, process, and analyze each file format.
2. Related data stored in separate files must be loaded to make comparisons.
3. Files require additional software to manage data access and record data edits.
4. One file cannot be used and modified by more than one user at a time.
5. Most file formats do not support data indexing which can improve data access and analysis performance.

A relational database management system (RDBMS) provides solutions to all of the disadvantages of file based storage. With data from many files stored in a database the power of the structured query language (SQL) can be used to simplify data access and analysis. Consider the following question a data user may ask, *for CReSIS data frames*

20091224_01_001 to 003 what is the average ice surface elevation? Let's examine this task using file based storage and database storage. Using data stored in MAT files, the following steps outline the process required to complete the task:

1. Search for files on the file system matching the frames in question
2. Load the matching files
3. Extract ice thickness from each file and combine the values
4. Calculate the average of the combined values

If the same data is stored in related tables in a database the task can be simplified into a single step: execute an SQL query to select and calculate the average ice thickness. Data in a database can take advantage of various indexing methods which allow SQL to quickly search for and find data resulting in a significant performance gain, even for this simple task. Code for both MATLAB (file based storage) and SQL (database storage) are shown in Fig. 9. The MATLAB method took 23.39 seconds to execute while the SQL method took only 0.25 seconds.

```
% FIND THE FILES THAT MATCH THE REQUIRED DATASET
baseFilePath = 'Z:\mdce\cr1\rds\2009_Antarctica_TO\CSARP_post\CSARP_layerData';
layerDataFns = get_filenames(baseFilePath,'Data_20091224_01','','.mat','recursive');

% PARSE THE FILES THAT MATCH THE REQUIRED FRAMES
outTwtt = [];
for fnsIdx = 1:3
    layerData = load(layerDataFns{fnsIdx}); % LOAD THE FILE
    outTwtt = cat(2,layerData.layerData{1}.value{2}.data); % STORE THE OUTPUT
end

% CALCULATE THE AVERAGE SURFACE
fileAvgSurfaceTwtt = mean(outTwtt);
```

A

```
SELECT avg(twtt) FROM rds_layer_points AS lp
    JOIN rds_point_paths AS pp ON lp.point_path_id = pp.id
    WHERE lp.layer_id = 1 AND pp.frame_id IN
        (SELECT id from rds_frames WHERE name
            IN ('20091224_01_001','20091224_01_002','20091224_01_003'));
```

B

Fig. 9: Code required to calculate the average ice thickness over a given set of CReSIS frames.

MATLAB code for file based data (A) and SQL for database data (B) are shown

4.3 Project Objectives

It should now be clear what problems were faced by CReSIS prior to the integration of the OPS. Given that current methods readily available to CReSIS for data storage and distribution are inadequate, the primary goal of the OPS project is to develop a free and open source SDI capable of storing, managing, creating, analyzing, and distributing the dataset collected by CReSIS in a way that provides an improved experience for both end users and the primary producers of polar remote sensing data. From this goal a set of clear objectives can be constructed for the OPS project:

1. Develop and deploy a database management system.
2. Develop and deploy a web-based data retrieval system (geoportal).
3. Develop and deploy an API for interaction between MATLAB and the new system.
4. Deploy the new system to the public and include community-derived datasets.
5. Share the source code following standard free and open source software (FOSS) guidelines.

The following sections outline the specific solutions developed and implemented by the OPS to achieve these goals.

5 OpenPolarServer System Structure and Components

5.1 Spatial Data Infrastructures

Before exploring the conceptual structure and detailed design of the OPS an overview of some definitions of spatial data infrastructures (SDI) is required. A SDI provides data users access to spatial data stored on remote servers [21]. In addition, a SDI allows for a single hub of data management for many data sources resulting in a reduction of the overall effort required to create and manage data [22][23]. "A SDI should enable the discovery and delivery of spatial data from a data repository, ideally via one or more web services. Additionally, it is often desirable that the data provider is able to (remotely) create and update spatial data stored in a repository. Hence, the basic software components of a SDI consist of: (i) a software client that can display, query, and analyze spatial data; (ii) a catalogue service for the discovery, browsing, and querying of metadata or spatial services, spatial datasets and other resources; (iii) a spatial data service that enables the delivery of the data via the Internet, and/or processing services such as datum and projection transformations; (iv) a (spatial) data repository, and (v) GIS software (client or desktop) that permits the creation and maintenance of data." [24]

The OPS SDI is a collection of software and custom code used to store, update, analyze, manage, and distribute data. A simplified SDI definition can be proposed: An SDI is the collection of software, hardware, and code needed to provide basic data services such as storage, management, analysis, and distribution to both data users and data producers. What software, hardware, and code is needed will vary based on the data, management, and distribution requirements. A conceptual outline of the components required for the OPS SDI

follows.

5.2 OpenPolarServer System Structure

Fig. 10 shows the conceptual structure of the OPS SDI. The SDI is composed of 7 basic components, outlined in sections 5.2.1 through 5.2.7. Each section outlines the basic function of the component and how it interacts with the rest of the SDI. Chapter 6 presents the actual software chosen to fill the roles of the components presented in this chapter.

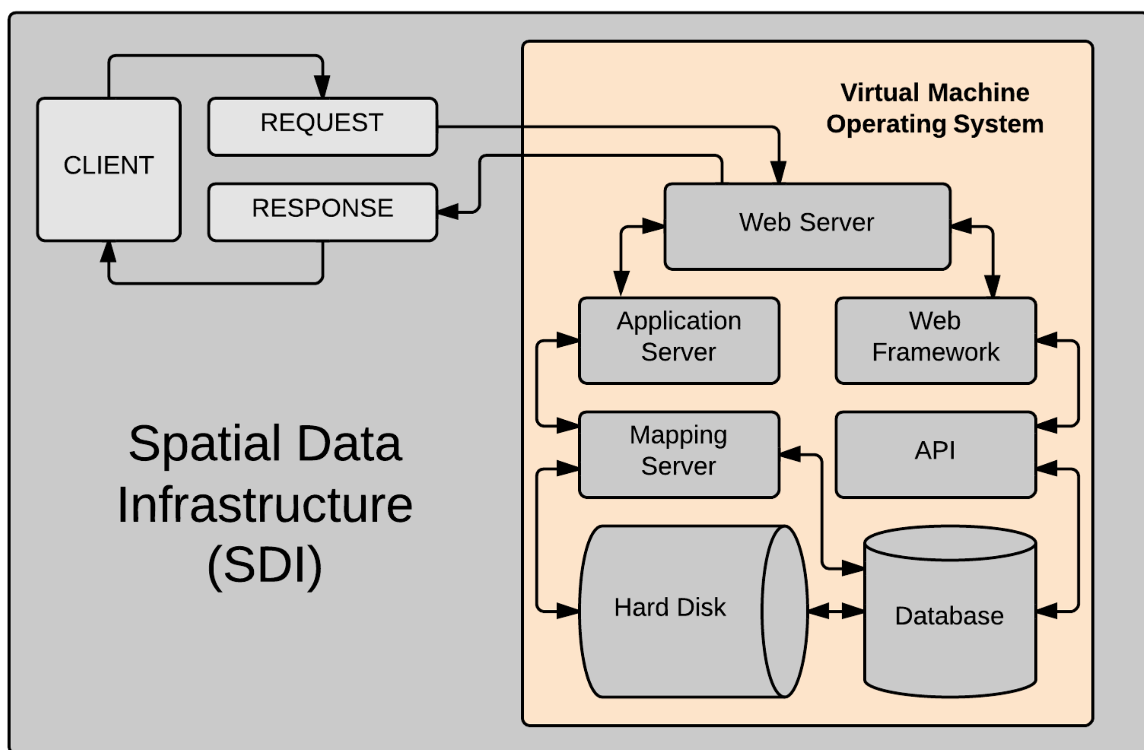

Fig. 10: Conceptual diagram of the OPS SDI components.

5.2.1 Virtual Machine (VM) and Operating System (OS)

The first component is the OS of the SDI. The OS (Windows, Linux, OSX, etc…) serves as the platform to install and execute other SDI components on the server. Often server OSs are installed on a VM. A VM is a software implementation of a computer. Using a VM allows a single set of physical server hardware (hard drive, random access memory, processors) to be shared by many separate VMs, each allocated a certain portion of the real hardware. The VM is shown by the tan colored area in Fig. 10. Note that all of the SDI components are within the VM installed on the OS of the SDI.

5.2.2 Relational Database Management System (RDBMS)

A SDI by definition does not require a RDBMS, but most SDIs include some form of database. A database is an organized collection of data [25]. This means that while data is stored ultimately on the hard drive the data is organized and managed as a collection of related tables by the RDBMS. A RDBMS has many advantages over hard drive only-storage including data management (access restrictions, data constraints), concurrency (multi-user data access), and indexing (efficient data access). A RDBMS supports the management of spatial data and is often the primary storage mechanism of SDIs.

5.2.3 Web Server

A key component in any SDI is a web server. A web server allows and controls the flow of data from anyone connecting to the server via the Internet. To access any of the other components of a SDI (application server, mapping server, database, web framework, etc…) a request must be sent to a web server by a client. The web server will process the request, and

pass it on to the appropriate SDI components for additional processing. On completion of a certain task a response will be returned from the SDI component to the web server which will then present the response to the client.

5.2.4 Web Application Server

A Web Application Server (WAS) is a special type of web server. Often applications running on the server (web applications) must be exposed to the web in order to make them accessible to users. A standard web server has no way of managing those server applications. The WAS stores and manages one or more server applications and exposes the applications functionality to clients by telling the standard web server that a web application is available. A client can then make requests to the web server which passes the request to the application hosted by the WAS. The web application will process the request (generating some result) and that result will be returned by the WAS to the web server and then returned to the client. The web server only handles the request/response objects and allows the WAS (and web applications) to handle the logic of the requests and generation of the responses.

5.2.5 Mapping Server

A Mapping Server is one of the web applications managed by a WAS. The mapping server is designed to generate a variety of georeferenced spatial data formats for return via standard web services. The mapping server serves spatial data from both the hard drive and database to the internet (via the WAS) through standard services such as web map service (WMS) [26], web feature service (WFS) [27], web map tile service (WMTS) [28], and others. The OPS primarily uses the WMS service, which is a protocol that defines request and response structures. A WMS response is a georeferenced image in a format such as JPEG or PNG. A quality open source mapping server typically implements open geospatial consortium

(OGC) [29] compliant services because most web clients will expect a response formatted in the OGC standard.

5.2.6 Web Framework

It is often desirable for a SDI to provide some custom data input/output and analysis services to clients. Those services can be created as custom web applications managed by the WAS as is the case of the mapping server. Another approach is to use a web framework. A web framework typically connects server-side code to the Internet via the web server. Often web frameworks also provide code libraries for accessing data in a database and generally provide code used to decode/encode request/response objects to and from the web server. Most complete web frameworks provide the scaffolding (basic functionality) and the developer then writes the code required for the data specific input, output, and analysis. It is due to the provided scaffolding that it is often desirable to use a web framework instead of creating a custom web application from scratch.

5.2.7 Application Programming Interface (API)

The custom code written with the web framework is generally referred to as the server side application programming interface (API). The API can be thought of as the brain (logic) of the OPS SDI where specific data input, output, and analysis functions are carried out on the server. Without APIs or web applications a web server can do nothing but serve files.

5.2.8 OPS Client Communications

The OPS SDI supports a wide variety of clients and request types. A client is simply any software or service that makes a request to the OPS web server. Some examples include a web browser, desktop application, or custom script in a language that supports web requests. The primary OPS clients include a web GeoPortal and a MATLAB application which are

discussed in the next chapter. Clients can make a variety of requests to the OPS SDI; the most common requests are shown in Fig. 11. A standard request (A) is best described by the process of visiting a webpage on the server:

1. A user enters a URL in a web browser, the browser sends a request to the web server.
2. The web server processes the request, finds the HTTP file and returns it.
3. The browser receives the response and renders the HTTP file.

An API request (B) is a process in which a user wants to receive the result of some process (code) defined on the server based on their inputs. This request is typically generated automatically by a client based on user inputs to a GUI but can also be generated by a user created script. The typical process of this kind of request follows:

1. A user (via a custom script or a GUI element) submits a request including some input data.
2. The client (which the user is interacting with) generates a JSON/XML string from the user's input and sends a request to the web server.
3. The web server processes the request, and passes on the input to the web framework.
4. The web framework executes some code (based on the user input) and generates a JSON/XML string and returns a response to the web server.
5. The web server returns a response to the client which handles the returned data and presents it to the user.

A map request (C) is the final type of request made by clients to the OPS SDI. The structure of this request is very similar to the API request (B). The typical process of this kind of request follows:

1. A user (almost always via a GUI element in a client) submits a WMS/WFS request to the web server.
2. The web server passes this request to the web application server, which passes the request

   to the web application (mapping server).
3. The mapping server processes the request and generates an image (JPEG/PNG) or data (XML/GML) and returns it to the web application server.
4. The web application server returns the response to the web server.
5. The web server returns the response to the client which handles the returned data and presents it to the user.

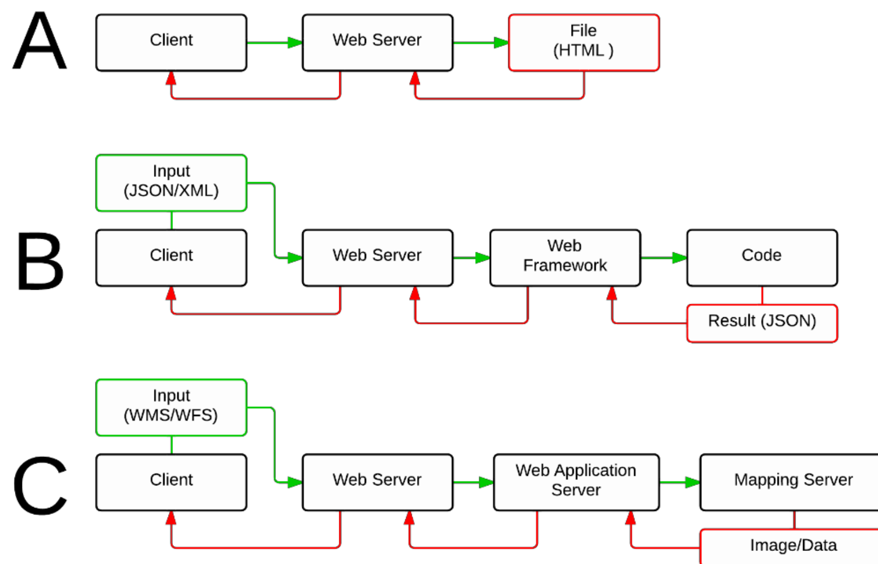

Fig. 11: Common web requests made by clients to the OPS server: a basic request (A), API request (B), and Map request (C).

6 OpenPolarServer SDI Implementation

The implementation of the OPS SDI follows the conceptual structure of the previous chapter exactly. This chapter presents, for each conceptual component, the actual software selected and its implementation on the OPS (Fig. 12).

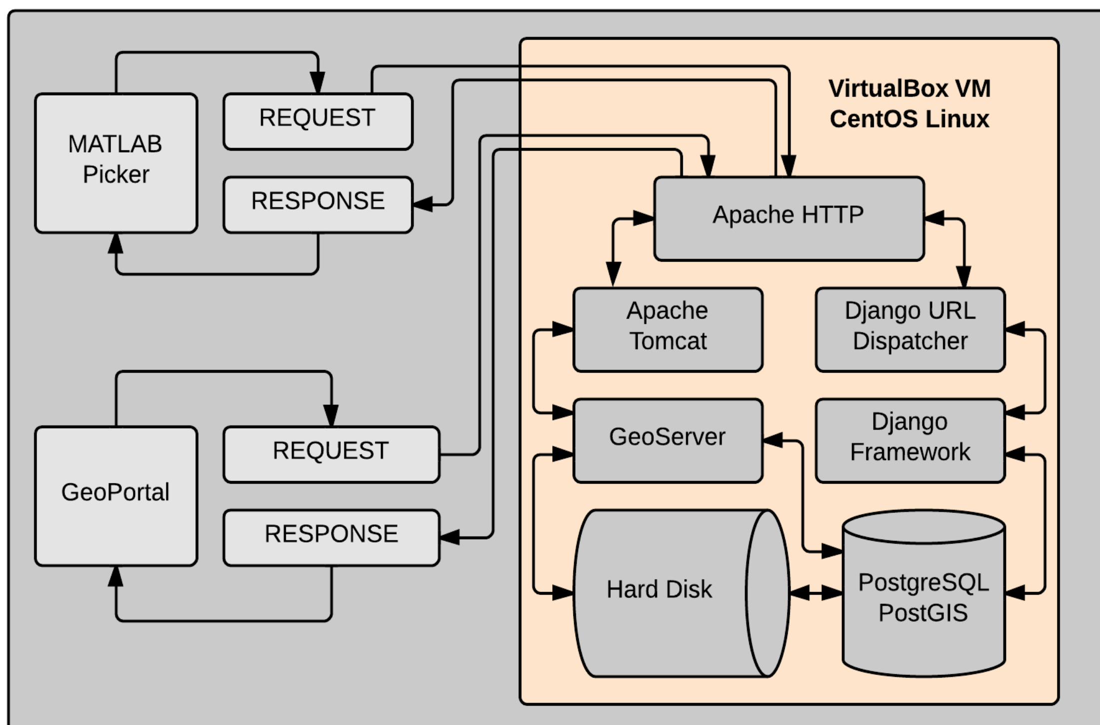

Fig. 12: A modified version of Fig. 10 showing the software selected to fill each SDI component role.

6.1 Free and Open Source Software (FOSS)

For all categories of GIS software required for the implementation of the OPS SDI, a free and open product is available [24]. In addition, a requirement of the OPS is that it can be constructed using only FOSS because of a limited software budget. Steiniger and Hunter [23] presented a five-step guide to selecting FOSS for a research project, which includes: (i) develop software use cases for your own context; (ii) establish a set of evaluation criteria based on the use cases; (iii) perform the software evaluation with respect to the established criteria; (iv) develop a weighting criteria according to application context; and (v) select software based on the results of the evaluation and weighting scheme. An attempt was made to follow this guide, but it often became clear that the specific FOSS software packages were the obvious (and sometimes only) choice in their categories.

6.2 CentOS Linux

The CentOS Linux [30] distribution is a stable, predictable, manageable and reproducible platform derived from the sources of Red Hat Enterprise Linux (RHEL) [31]. The CentOS Project is a community-driven free software effort focused around the goal of providing a rich base platform for open source communities to build upon [30]. This Linux distribution was selected because it is open source and based on the commonly used RHEL. CentOS Linux also has support for all of the required software and tools that were chosen and described in the following sections. Software can be installed on the CentOS Linux OS using the YUM package manager [32] and the RedHat Package Manager (RPM) [33].

6.3 PostgreSQL and PostGIS

One of the major contributions of the OPS SDI to CReSIS is the use of a relational database management system (RDBMS) for the storage of previous standard data files. The

RDBMS chosen is PostgreSQL, a powerful, open source object-relational database system [34]. PostGIS was chosen as the spatial database extender for PostgreSQL and adds support for spatial queries in SQL [35]. In addition, the psycopg2 Python module is installed and used to communicate with the database (PostgreSQL/PostGIS) through Python.

```
1   # INSTALL THE PGDG REPO
2   wget http://yum.postgresql.org/9.3/redhat/rhel-6-x86_64/pgdg-centos93-9.3-1.noarch.rpm
3   rpm -Uvh pgdg-centos93-9.3-1.noarch.rpm
4
5   # EXCLUDE POSTGRESQL FROM THE BASE CentOS RPM
6   sed -i -e '/^\[base\]$/a\exclude=postgresql*' /etc/yum.repos.d/CentOS-Base.repo
7   sed -i -e '/^\[updates\]$/a\exclude=postgresql*' /etc/yum.repos.d/CentOS-Base.repo
8
9   # INSTALL POSTGRESQL
10  yum install -y postgresql93* postgis2_93*
11
12  # INSTALL PYTHON PSYCOPG2 MODULE FOR POSTGRES
13  export PATH=/usr/pgsql-9.3/bin:"$PATH"
14  pip install psycopg2
```

Fig. 13: Installation of PostgreSQL, PostGIS, and Psycopg2 on CentOS Linux.

This software is installed using YUM and the PostgreSQL PDGD RPM repository. Lines 1-7 in Fig. 13 show the inclusion of the PDGD RPM (not included by default with CentOS) and the exclusion of the CentOS standard PostgreSQL package. An RPM repository is simply a directory of software packages hosted and managed on the web. Including an RPM repository in Linux allows the YUM package manager to access and install software from the RPM directory. Using the PDGD RPM ensures the most recent PostgreSQL and PostGIS software updates. Line 10 shows the YUM installation of PostgreSQL 9.3 and PostGIS2. Finally lines 13-14 show the installation of the psycopg2 Python to PostgreSQL

adapter.

After PostgreSQL and PostGIS are installed there is some initial configuration that must be completed before using the database management system. The configuration consist of the initiation of a database server, creation of an admin user, creation of a PostGIS template database and finally the creation of the actual OPS database. All of these steps are completed using the PL/pgSQL SQL procedural language and pgsql interactive terminal. At the completion of the code (Fig. 14) the OPS has a running PostgreSQL server with an empty database created and stored on the CReSIS networked hard drives. Placing the database on network drives allows the OPS system to leverage the existing CReSIS incremental backup system. As data is updated in the OPS database, the CReSIS network drives are backed up to tape or discs. While PostgreSQL does offer many backup and replication services, natively leveraging the existing CReSIS backup infrastructure saved time and provides the same essential function as the native PostgreSQL backup services.

```
1   # INITIALIZE THE DATABASE CLUSTER
2   pgDir='/cresis/snfs1/web/ops/pgsql/9.3/'
3   cmdStr='/usr/pgsql-9.3/bin/initdb -D '$pgDir
4   su - postgres -c "$cmdStr"
5   pgConfDir=$pgDir"postgresql.conf"
6
7   [...]
8
9   # START UP THE POSTGRESQL SERVER
10  su - postgres -c '/usr/pgsql-9.3/bin/pg_ctl start -D '$pgDir
11  sleep 5
12
13  # CREATE THE ADMIN ROLE
14  cmdstring="CREATE ROLE "$dbUser" WITH SUPERUSER LOGIN PASSWORD '"$dbPswd"';"
15  psql -U postgres -d postgres -c "$cmdstring"
16
17  # CREATE THE POSTGIS TEMPLATE
18  cmdstring="createdb postgis_template -O "$dbUser
19  su - postgres -c "$cmdstring"
20  psql -U postgres -d postgis_template -c "CREATE EXTENSION postgis; CREATE EXTENSION
    postgis_topology;"
21
22  # CREATE THE APP DATABASE
23  cmdstring="createdb "$dbName" -O "$dbUser" -T postgis_template"
24  su - postgres -c "$cmdstring"
```

Fig. 14: Configuration of PostgreSQL and PostGIS on CentOS Linux

The complete OPS database schema is shown in the form of an entity-relationship (ER) diagram (Fig. 15). To create this schema a standard process of database design was followed, which includes 8 steps: (1) Determine the purpose of the database, (2) Find and organize the information required, (3) Divide the information into tables, (4) Turn information items into columns, (5) Specify primary keys, (6) Set up the table relationships, (7) Refine the design, and (8) Apply the normalization rules [36]. This is an iterative process and steps 7 and 8 were repeated many times to fine-tune the database design. The major goal of the database design process is to improve query efficiency while minimizing data redundancy in the database. More about the actual creation of the schema in the PostgreSQL

software will be discussed in section 6.6.3.

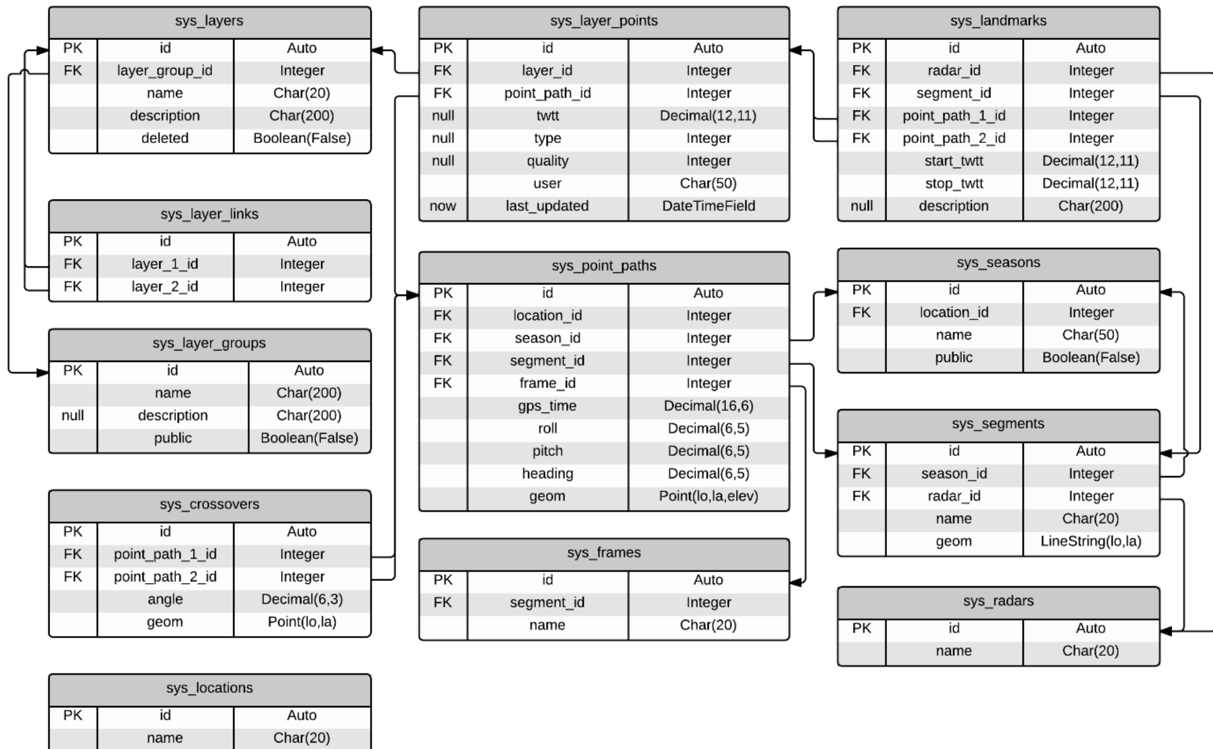

Fig. 15: The OPS Database ER (Entity-Relationship) diagram.

6.4 Apache HTTP and Apache Tomcat

To allow external users, software, and APIs to access the components on the OPS SDI, a web and web application server are necessary. The Apache Software Foundation [37] products HTTP and Tomcat were chosen as the web and web application server, respectively. The Apache Software Foundation has been the go-to source for open source web servers and more since their creation in 1999. Because the HTTP server and Tomcat server are both open source and industry accepted they were chosen for the OPS. The Apache HTTP web server and Tomcat web application server are installed on CentOS Linux

using the YUM package manager and the command *yum install httpd tomcat6*.

```
<VirtualHost *:80>

    ServerAdmin "$serverAdmin"
    DocumentRoot /var/www/html
    ServerName "$serverName"

    ErrorLog /var/www/sites/"$serverName"/logs/error_log
    CustomLog /var/www/sites/"$serverName"/logs/access_log combined
    CheckSpelling on

    ScriptAlias /cgi-bin/ /var/www/"$serverName"/cgi-bin/
    <Location /cgi-bin>
        Options +ExecCGI
    </Location>

    Alias /data "$webDataDir"
    <Directory "$webDataDir">
        Options Indexes FollowSymLinks
        AllowOverride None
        Order allow,deny
        Allow from all
        ForceType application/octet-stream
        Header set Content-Disposition attachment
    </Directory>

    Alias /profile-logs /var/profile_logs/txt
    <Directory /var/profile_logs/txt>
        Options Indexes FollowSymLinks
        AllowOverride None
        Order allow,deny
        Allow from all
    </Directory>

</VirtualHost>
```

Fig. 16: Apache HTTPD primary server configuration file.

The primary configuration file of the Apache HTTP web server (httpd) is shown in Fig. 16. The entire server is running as a virtual host on port 80 (see lines 1 and 34), port 80 is the standard HTTP web port. On the CReSIS hosted OPS this is changed to port 443, the standard HTTPS (secure HTTP) web port. Lines 3-5 set up some basic parameters of the web server including the server administrator (an email), document root (the default web directory on the server hard drive), and the server name (an IP address or web domain name). Lines 7-9 set up

error logging and access logging for the server and enable case insensitive file searching. For example if there is an HTML file called *about.html* but the client (user) enters *About.html* the Apache web server will find the actual file instead of returning a 404 (not found) error. Lines 11- 14 configure a CGI Proxy that is used by the OpenLayers JavaScript library to make XMLHttpRequests by avoiding security restrictions in native JavaScript [38]. Lines 16-24 configure a directory for serving data files generated by the server side API. The configuration in these lines force files in the server data directory $*webDataDir*, mapped to the URL **/data*, to auto-download when requested. Finally lines 26-32 configure another web accessible download directory mapping the server directory */var/profile_logs/txt* to the URL **/profile-logs* which stores txt files generated during automated code profiling (code performance monitoring). Generally the role of <*directory*> tags is to configure access to a server directory (such as

/var/profile_logs/txt) to be accessible at a web directory such as *http://[ServerName]/profile-logs.*

6.5 Geoserver

One of the requirements of the OPS SDI is that it be able to distribute geographic data in the form of georeferenced map images to the clients discussed in section 6.7. To achieve this the GeoServer web application was selected. GeoServer is an open source software server (web application) that shares geospatial data using open standards [39]. GeoServer was chosen for the OPS because it is free and open source and is the reference implementation for the OGC web services standards. The role of GeoServer on the OPS SDI is to take spatial data from the server hard drive and database and generate georeferenced map images.

GeoServer can be installed in many different ways, including as its own complete standalone web/application server. Because the OPS SDI already has a web server for other purposes the GeoServer WAR (web archive) installation method was selected. Using the WAR method means that GeoServer is deployed as a JAVA web application managed by the Apache Tomcat web application server. Deploying an application in WAR format on Apache Tomcat is a very simple process. A file *geoserver.war* is downloaded and placed in the Tomcat web application directory. On CentOS Linux this directory is */var/lib/tomcat6/webapps/*. With a WAR file (just a standard zip file with a .war extension instead of .zip) placed in the web application directory the first time the Tomcat server is started the WAR file will be unpacked (unzipped) into a data directory. The data directory (at least for the GeoServer web application) contains all of the configuration files for data, security, formats, and styles.

Often a static external data directory is used to make upgrading the GeoServer WAR application simple. Fig. 17 shows the installation of the *geoserver.war* and the configuration of an external GeoServer data directory. Keeping a data directory (which contains all of the OPS GeoServer configuration) external to the GeoServer WAR allows a developer to simply place a new *geoserver.war* file in the Tomcat web application directory each time there is a software upgrade. A Linux environment variable *GEOSERVER_DATA_DIR* is set (Fig. 17 line 2-3) that tells the GeoServer web application (installed on line 6) where to look for the data directory. Ownership and permissions are set for the data directory on lines 9-10 and finally the Tomcat web application server is started (deploying *geoserver.war* automatically) on line 13.

The final step in the installation of GeoServer is to set up a proxy that tells the Apache HTTP web server that the web application is available from Apache Tomcat. The standard web

```
# WRITE ~/.bashrc ENVIRONMENT VARIABLES
echo 'GEOSERVER_DATA_DIR="/cresis/snfs1/web/ops/geoserver"' >> ~/.bashrc # GEOSERVER
. ~/.bashrc # RELOAD VARIABLES

# COPY THE GEOSERVER WAR TO TOMCAT
cp /vagrant/conf/geoserver/geoserver.war /var/lib/tomcat6/webapps

# SET OWNERSHIP/PERMISSIONS OF GEOSERVER DATA DIRECTORY
chmod -R u=rwX,g=rwX,o=rX /cresis/snfs1/web/ops/geoserver/
chown -R tomcat:tomcat /cresis/snfs1/web/ops/geoserver/

# START APACHE TOMCAT
service tomcat6 start
```

Fig. 17: GeoServer installation as a Tomcat6 web application on CentOS Linux.

application port for Apache Tomcat is 8080. Lines 9-10 in Fig. 18 tell Apache HTTP to proxy (or forward) request of the URL http://[SomeServer]*/geoserver* to the geoserver application running at *http://localhost:8080/geoserver*.

```
ProxyRequests Off
ProxyPreserveHost On

<Proxy *>
    Order deny,allow
    Allow from all
</Proxy>

ProxyPass /geoserver http://localhost:8080/geoserver
ProxyPassReverse /geoserver http://localhost:8080/geoserver
```

Fig. 18: Proxy configuration telling HTTP that GeoServer is available from Tomcat.

Once the *geoserver.war* is installed and the proxy in place the OPS clients will be able to access all of the features of the GeoServer web application by making requests to http://[SomeServer]/geoserver. However, without some additional configuration specific to the OPS data there will be nothing for them to request. GeoServer configuration is done using a built-in web administration interface (Fig. 19). The configurations shown in the interface are stored in XML files in the GeoServer data directory. Again, the use of an external OPS GeoServer data directory means that configuration will stay in place each time the application is upgraded. The first time GeoServer is set up some initial configuration must be done including the creation of users, passwords, security settings, and more. Details on these configurations are available in the GeoServer online documentation.

Data in GeoServer is organized into workspaces, stores, and layers (Fig. 20). A workspace is simply a group of related stores. There is no requirement for the type of grouping a workspace represents. The OPS GeoServer contains two workspaces *arctic* and *antarctic*. Again, workspaces do not needed to be organized in any particular way and the location-based organization is just what worked best for the data included in the OPS GeoServer. Each workspace contains stores, which are containers of data, some examples being a shapefile, GeoTIFF, directory of shapefiles, or PostGIS database. Finally each store contains one or many layers. A shapefile or GeoTIFF store will contain a single layer (since it is a single file) but stores such as a PostGIS database or directory of shapefiles can contain many layers. Configuration such as projection, styling, and bounding boxes are set at the layer level.

Fig. 19: GeoServer Web Administration Interface

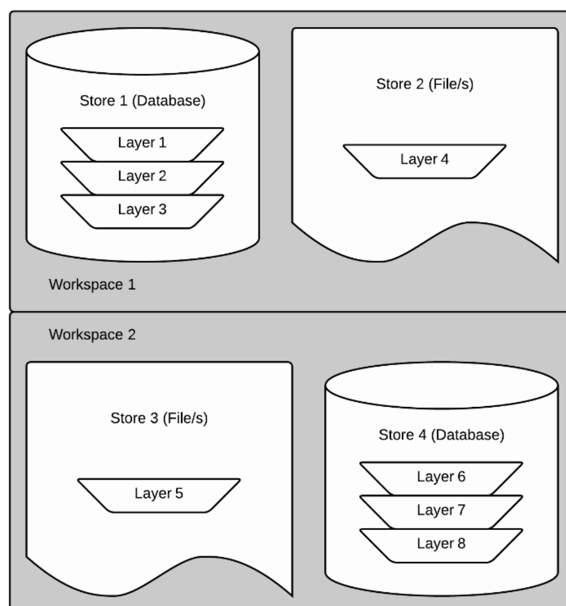

Fig. 20: Conceptual relationship between GeoServer workspaces, stores, and layers.

Once a layer is configured and published by GeoServer (meaning at least one store and workspace have also been created and published) the data is available for preview using the GeoServer built-in layer preview tools. Using these tools you can preview the published layer in any of the GeoServer output format types [40]. Fig. 21 shows a GeoServer layer preview in the OpenLayers output format (a WMS format) of Natural Earth GeoTIFF data. The URL (split onto multiple lines for clarity) used to generate the preview is also shown.

```
1   http://ops.cresis.ku.edu/geoserver/arctic/wms?service=WMS
2   &version=1.1.0
3   &request=GetMap
4   &layers=arctic:arctic_naturalearth
5   &styles=
6   &bbox=-8125549,-6101879,8186727,3197247
7   &width=578
8   &height=330
9   &srs=EPSG:3413
10  &format=application/openlayers
```

A

B

Fig. 21: GeoServer layer preview (OpenLayers format) of Natural Earth data (B) and the request URL used to generate the preview (A).

The GeoServer PostGIS data store offers a layer type which is used by the OPS called an SQL view. This layer type allows you to write SQL which is executed each time the layer is requested returning a geometry object from a PostGIS database. The SQL view layer type (and PostGIS store type) are used to distribute CReSIS L2 flight path data stored in the

PostgreSQL/ PostGIS database as map images. A useful feature of the SQL view is that it can be parameterized. This means that custom input variables can be added to the request URL which are input into the SQL that is executed when the layer is requested. This makes the layer dynamic in the sense that an image showing different data can be requested each time (unlike a static GeoTIFF layer.

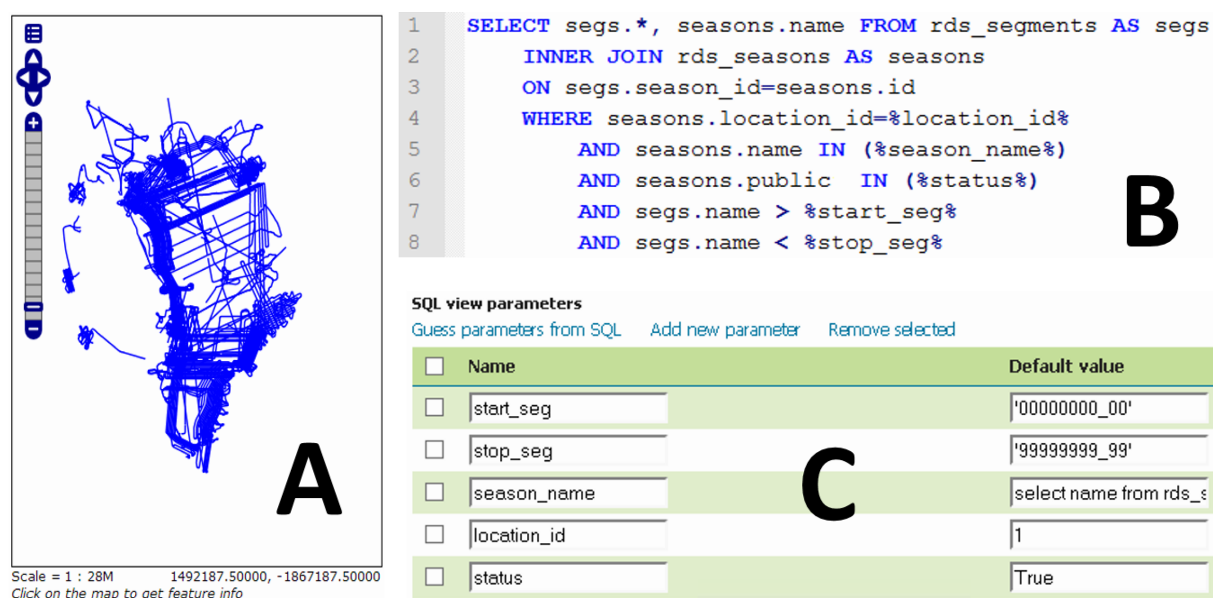

Fig. 22: Configuration for a GeoServer SQL view layer. A layer preview (OpenLayers format) (A), and SQL code for the view (B), and SQL view parameters (C) are shown.

Configuration for the CReSIS radar depth sounder L2 flightlines SQL view and a preview over Greenland are shown in Fig. 22. Note that default values are set for all of the SQL view parameters (C) so if a client does not include any input values all of the defaults are

used. The request URL (Fig. 23) for the SQL view layer is very similar to the request for the GeoTIFF layer (Fig. 21). In fact if you exclude line 11 it is an identically formatted request. Line 11 adds a tag called *viewparams* and specifies two parameters *start_seg='2010101_00' and stop_seg='20121231_99'*. The *viewparams* tag is used to pass named inputs into an SQL view. The named inputs must be defined SQL view parameters in the SQL view configuration. This example will limit the result to CReSIS data segments that fall in between 20100101_01 and 20121213_99. Note that the *viewparams* tag can be completely excluded (to accept all the default values) or can be included with any number of the configured SQL view parameters accepting defaults for the any that are excluded.

```
1  http://ops.cresis.ku.edu/geoserver/arctic/wms?service=WMS
2  &version=1.1.0
3  &request=GetMap
4  &layers=arctic:arctic_rds_line_paths
5  &styles=
6  &bbox=-1500000.0,-4000000.0,1500000.0,0.0
7  &width=384
8  &height=512
9  &srs=EPSG:3413
10 &format=application/openlayers
11 &viewparams=start_seg:'2010101_00';stop_seg:'20121231_99'
```

Fig. 23: Request URL for the arctic radar depth sounder CReSIS L2 flightlines GeoServer SQL view.

A final step in GeoServer configuration is the specification of styles using the styled layer descriptor (SLD) syntax which is just simple XML. Fig. 24 shows the SLD for a simple black 2 pixel wide line. There are many SLD styles included with GeoServer so the creation of custom styles is optional. More information on SLD is available in the GeoServer documentation [41].

```
1  <FeatureTypeStyle>
2      <Rule>
3          <Name>blackLine</Name>
4          <Title>Black Line</Title>
5          <Abstract>A solid black line with a 2 pixel width</Abstract>
6          <LineSymbolizer>
7              <Stroke>
8                  <CssParameter name="stroke">#000000</CssParameter>
9                  <CssParameter name="stroke-width">2</CssParameter>
10             </Stroke>
11         </LineSymbolizer>
12     </Rule>
13 </FeatureTypeStyle>
```

Fig. 24: SLD (Styled Layer Descriptor) for a simple black line.

6.6 Django

Django [42] is a Python Web framework that offers a complete Python library used to connect clients to the OPS server side database and custom server-side code. Django follows the “don’t repeat yourself (DRY)” philosophy, which the OPS tries to follow as well. Using Django features, such as the Object-Relational mapper, URL dispatcher, Database-Access API, and more, the OPS system is able to leverage an existing and tested library to complete many of the commonly required tasks of a SDI. The following sections discuss Django and the OPS implementations of many of the Django features.

6.6.1 Installation and Configuration

Installing the Django web framework on CentOS Linux is very simple. The YUM package manager can be used to install the Python PIP (python indexing project) package manager which allows the installation of any Python module (including Django) with a command of the form *pip install [module name]*. This is the standard process for installing Python modules on CentOS. Fig. 25 shows the installation and configuration of both the Django Python module and the OPS Django project. Lines 2 and 5 install pip (using the YUM package manager) and Django (using the pip package manager). Lines 8-9 copy an existing Django project (the OPS Django project) into the standard Django directory on the server.

```
1   # INSTALL PIP
2   yum install python-pip
3
4   # INSTALL DJANGO
5   pip install Django==1.6.2
6
7   # CREATE DIRECTORY AND COPY PROJECT
8   mkdir -p /var/django/
9   cp -rf /vagrant/conf/django/* /var/django/
10
11  # SYNC THE DJANGO DEFINED DATABASE
12  python /var/django/$appName/manage.py syncdb --noinput
```

Fig. 25: CentOS Linux installation of the Django Python module and OPS Django project.

A Django project contains all of the configuration, data models, and custom code required for the Django module to operate. The Django Python module offers all of the functionality but the files inside of the Django project define the actual operation of Django for the OPS server. Django offers a management tool *manage.py* which allows a developer to create a recommended scaffolding (empty directory structure and files) for a Django project. This was completed during the initial development of the OPS. A slightly modified Django project structure (Fig. 26) was selected to best serve the needs of the OPS SDI. The biggest change is that in the default Django project structure each application (a sub directory of a project) has its own *views* file. This file contains the custom code that can be executed by requests to the server. In the OPS Django project the *views* code is common to all applications and therefore is stored at the project level. Some unused files and directories were excluded and some new files *utility, authip, and monitor* were added to support various features needed

by the OPS SDI. The function of these files will be discussed in the following sections.

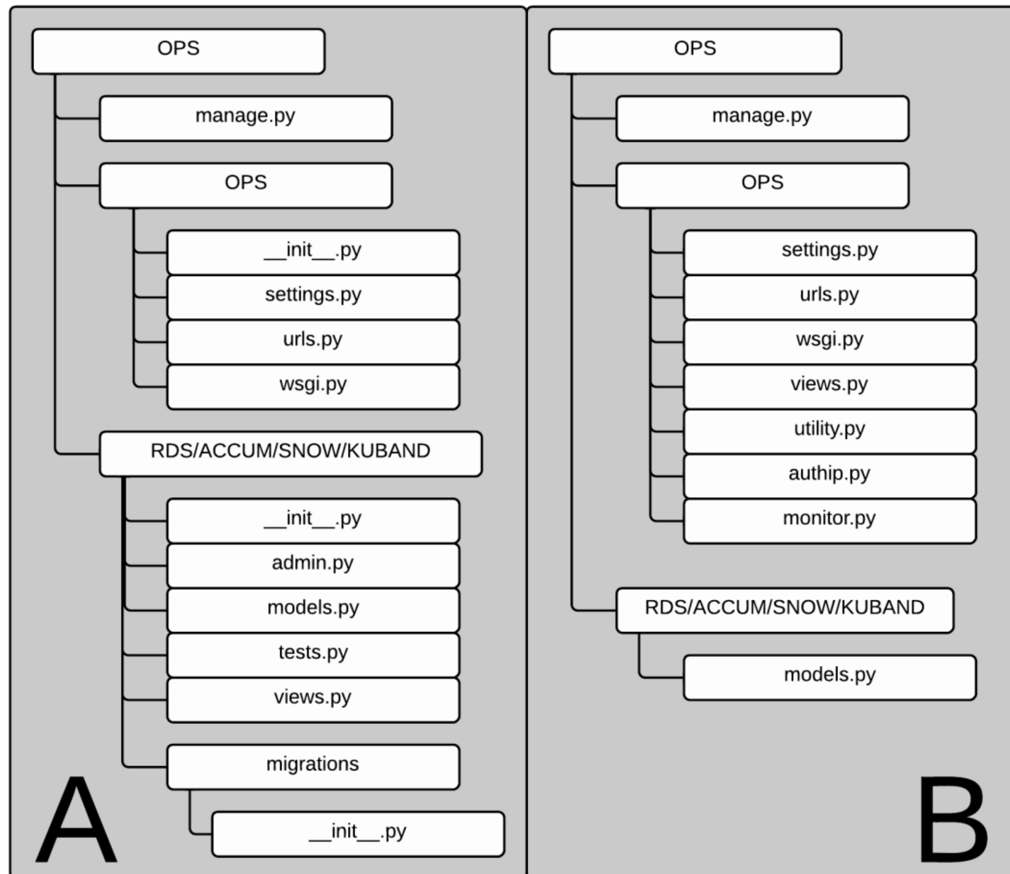

Fig. 26: An example of the default layout of a Django project (A) and the actual OPS Django project layout (B).

6.6.2 Model-View-Controller (MVC)

The Django project is implemented using the MVC concept. MVC is a pattern for organizing software into three related parts: models (M), views (V), and controllers (C). In the typical definition of MVC a model contains definitions of data, rules, logic, and functions, a view is the definition of an output or representation of data (such as a chart, diagram, or page), and a controller takes input from a user and passes it to a model [43]. The use of MVC in a web application context will be discussed in section 6.7.2. Django has a slightly different perspective on the MVC concept [44]. Django defines the models as Python classes which represent the schema of a database, views as custom callbacks (for rendering of any output type), and defines the controller as the framework itself. The framework provides methods for getting input from a user (generally from a web server) and passing that input to views. It also provides methods inside of views for accessing data in models and creating results from that data. Fig. 27 shows how the Django MVC is used to handle a standard web request. Each of the Django MVC components are discussed in detail in the following sections.

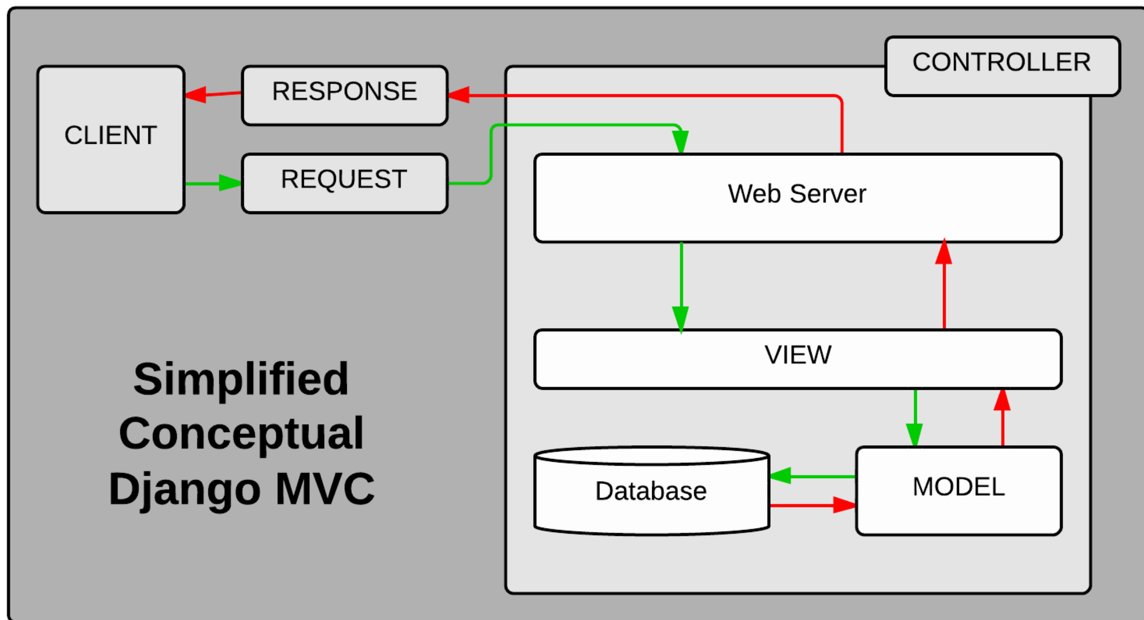

Fig. 27: A simplified diagram of the conceptual layout of the Django MVC framework in the context of a web request.

6.6.3 Database Models

In section 6.6.2 it was stated that Django defined the M (models) of MVC as Python classes which represent the schema of a database. These models are the primary objects which Django uses to perform callback functions created by developers (more on this in section 6.6.5). Each model (class) defines the name, fields, data types, and relationships of a single table in a database. The complete set of models, stored at the Django app level (Fig. 26) in the *models.py* file, define the entire schema of a database. These classes do not simply correspond to an already created database schema, they are in fact used by Django to generate the SQL required to create the schema in a database. Django offers support for many DBMS including PostgreSQL, MySQL, SQLite, and the Oracle Database Server. The Django file *settings.py* defines the database backend (PostgreSQL and PostGIS for the OPS) which Django uses to generate database specific SQL from the models. This means that the same models can be used to create a working database schema on any of the supported DBMS.

Fig. 28 shows two examples of Django models for the OPS database, the segments and frames models. The class definition defines the name of the table which will become *[app name]_[class name]* in the database. For example these models are part of the rds (radar depth sounder) Django app so the first model (lines A1-A10) will become a table called *rds_segments* in the OPS database. The creation of this table is shown in lines B1-B8. Part B is the Django generated SQL which creates the database schema defined by Django models on the PostgreSQL database. After a complete schema is created in *models.py* the Django file *manage.py* and the command *syncdb* can be used to generate the SQL (Fig. 28 B) for all of the project models and execute it on the database defined in *settings.py*.

```
1  class segments(models.Model):
2  
3      season = models.ForeignKey('seasons')
4      radar = models.ForeignKey('radars')
5      name = models.CharField(max_length=20)
6      geom = models.LineStringField()
7      objects = models.GeoManager()
8  
9      def __unicode__(self):
10         return "%s"%(self.name)
11 
12 class frames(models.Model):
13 
14     segment = models.ForeignKey('segments')
15     name = models.CharField(max_length=20)
16 
17     def __unicode__(self):
18         return "%s"%(self.name)
```

A

```
1  CREATE TABLE "rds_segments" (
2      "id" serial NOT NULL PRIMARY KEY,
3      "season_id" integer NOT NULL REFERENCES "rds_seasons" ("id")
4          DEFERRABLE INITIALLY DEFERRED,
5      "radar_id" integer NOT NULL REFERENCES "rds_radars" ("id")
6          DEFERRABLE INITIALLY DEFERRED,
7      "name" varchar(20) NOT NULL,
8      "geom" geometry(LINESTRING,4326) NOT NULL);
9  
10 CREATE TABLE "rds_frames" (
11     "id" serial NOT NULL PRIMARY KEY,
12     "segment_id" integer NOT NULL REFERENCES "rds_segments" ("id")
13         DEFERRABLE INITIALLY DEFERRED,
14     "name" varchar(20) NOT NULL);
```

B

Fig. 28: Django models (A) and the dynamically generated SQL for creating the models schema in a PostgreSQL database (B) for the segments and frames tables of the OPS database.

6.6.4 Database-Access API

Django includes an API (library of Python functions) for executing abstracted SQL on any database generated using the Django models. When a Django model is created it inherits the Django models class *Models.model* which includes all of the Django database-access class methods. These class methods allow a developer to do database CRUD (create, read, update, and delete) tasks. Behind the scenes the class methods dynamically generate SQL statements that are executed (using Psycopg2 for PostgreSQL) on the database. The database-access API also handles the opening and closing of database connections eliminating the need to write verbose and repetitive code to perform database CRUD tasks. The return from the SQL statement is encoded into a Python object called a Django QuerySet.

A

```
1  segmentsObj =
2  models.segments.objects.filter(
3      season__name__in=['2010_Greenland_DC8','2011_Greenland_P3'],
4      name__range=('20100000_00','20119999_99')
5  ).values_list('name','geom')
```

B

```
1  SELECT "rds_segments"."name",
2         "rds_segments"."geom"
3  FROM "rds_segments"
4  INNER JOIN "rds_seasons" ON ("rds_segments"."season_id" = "rds_seasons"."id")
5  WHERE ("rds_segments"."name" BETWEEN '20100000_00' AND '20119999_99'
6         AND "rds_seasons"."name" IN ('2010_Greenland_DC8','2011_Greenland_P3'))
```

Fig. 29: A database query made using the Django database-access API (A) and the actual SQL executed by the API in raw SQL form (B).

Fig. 29 shows a query on the segments table (defined by the segments model). Part A shows how the Django database-access API is used to perform SQL queries. The segments model *models.segments.objects* class is used to interact with the segments table in the database. The Django provided class method *.filter()* (inherited from *Models.model*) is used to perform an SQL SELECT query. The arguments passed to *.filter()* are table field names and the double- underscore is a special Django constructor which allows SQL methods such as IN, LIKE, and CONTAINS to be used. Note that the result from *.filter()* is a Django QuerySet which can be accessed like any standard Python object using the dot operator. However it is often desirable to request only a few values in an SQL query and store the results directly in a Python list. The Django method *.values_list()* forces the returned QuerySet object to be converted into Python lists. For the segments query example the result would be a list of the format *[[name1, ..., nameN],[geom1, ..., geomN]]* as the two requested output variables are the name and geom fields of the table. For all types of SQL queries there are Django methods that emulate there functionality including but not limited to *.get_or_create()* (select if exists else insert), *.delete()* (delete from), *.update()* (modify existing data).

The segments query example, when executed, automatically creates a database connection, generates the required SQL, executes the query, and encodes the response in Python all behind the scenes. This type of Django query is the basic logical element of a Django View.

6.6.5 Python Views API

A view in Django is simply a Python callback that takes some input, performs some logic (usually using the database-access API), and returns a formatted response object. The Python Views API is the brain of the OPS server. All of the CRUD tasks for CReSIS data are written as callbacks (views) in the Python Views API. In addition to the Django framework the views have access to any Python library installed on the system. Fig. 30 shows a complete list of the views created for the OPS. Since the complete OPS views API contains over 2000 lines of code it cannot all be included and explained here. All of the views follow the same basic pattern and only the internal logic of each callback is unique.

View Name	**View Description**
query	Executes any SQL query string
profile	Runs any of the following functions and returns a profiling log
analyze	Submits a PostgreSQL ANALYZE command for a set of tables
createPath	Creates a flightline path in the database
createLayer	Creates a layer in the database
deleteLayer	Deletes a layer from the database (sets status=Deleted for safety)
createLayerPoints	Creates layer points in the database
deleteLayerPoints	Deletes layer points from the database
deleteBulk	Deletes all related rows for a given set of inputs
releaseLayerGroup	Makes a group of layers public
releaseSeason	Makes a season public
getPath	Gets a flightline path from the database
getFrameClosest	Gets the closest frame to a point from the database
getLayers	Gets all the layers from the database
getLayerPoints	Gets layer points from the database
getLayerPointsCsv	Gets/Creates a CSV file of layer points from the database
getLayerPointsKml	Gets/Creates a KML file of layer points from the database
getLayerPointsMat	Gets/Creates a MAT file of layer points from the database
getLayerPointsNetcdf	Gets/Creates a NetCDF file of layer points from the database
getSystemInfo	Gets information on what data is in the database
getSegmentInfo	Gets information about a segment from the database
getCrossovers	Gets information about crossing flightlines from the database
getFrameSearch	Search's for and gets a frame from the database
getInitialData	Gets/Creates a zipped pack of data from the database

Fig. 30: A complete list of the OPS Python Views and a brief description of each.

In order to describe the implementation of a Django view the *getFrameClosest* view which finds the closest geometry (frame) from the frames table to an input point geometry will be detailed in full. Fig. 31 shows the function definition and python docstring. Note that the input to the function is a request. This request is an HTTP request object which was passed to the Apache HTTP web server by a user (client) and routed to this function by methods described in section 6.6.7.

```
1  def getFrameClosest(request):
2      """ Gets the closest frame from the OPS database.
3  
4      Input:
5          location: (string) region of frame to retrieve
6          x: (float) x value of point to find the closest frame to
7          y: (float) y value of point to find the closest frame to
8  
9      Optional Input:
10         season: (string or list of strings) season/s of frame to retrieve
11         startseg: (string) minimum segment name the output frame can belong to
12         stopseg: (string) maximum segment name the output frame can belong to
13         status: (boolean) can be used to get frames for private seasons
14 
15     Output:
16         status: (integer) 0:error 1:success 2:warning
17         data:
18             season: (string) the season name of the closest frame
19             segment_id: (integer) the segment id of the closest frame
20             start_gps_time: (float) the start gps time of the closest frame
21             stop_gps_time: (float) the stop gps time of the closest frame
22             frame: (string) the name of the closest frame
23             X: (list of floats) the x coordinates of the closest frame
24             Y: (list of floats) the y coordinates of the closest frame
25             gps_time: (list of floats) the gps times of the closest frame
26             echograms: (list of strings) urls of the ftp echograms of the closest frame
27 
28     """
```

Fig. 31: The function definition and docstring for the getFrameClosest Django view.

The request object contains a JSON string (see section 6.6.6) which contains the input data to the function provided by the user. Fig. 32 shows the first logical section of the function.

```
29      models,data,app = utility.getInput(request) # get the input and models
30
31      # parse the data input
32      try:
33          inLocationName = data['properties']['location']
34          inPointX = data['properties']['x']
35          inPointY = data['properties']['y']
```

Fig. 32: Getting function inputs and parsing the inputs for required data.

Here a utility function *getInput()* is called which returns the database models for the requested application, a Python object containing the decoded input values, and a string containing the application requested. Fig. 33 details the *getInput()* function and its depended functions.

```
1  def getInput(request):
2  
3      app,jsonData = getData(request)
4  
5      models = getAppModels(app)
6  
7      try:
8          data = json.loads(jsonData,parse_float=Decimal)
9      except:
10         return errorCheck(sys)
11 
12     return models,data,app
13 
14 def getData(request):
15 
16     if request.method == 'POST':
17         try:
18             app = request.POST.get('app')
19             data = request.POST.get('data')
20             if data is not None and app is not None:
21                 return app,data
22             else:
23                 response(0,'ERROR: INPUT VARIABLE ''data'' OR ''app'' IS EMPTY'),''
24         except:
25             response(0,'ERROR: COULD NOT GET POST'),''
26     else:
27         response(0,'ERROR: METHOD MUST BE POST'),''
28 
29 def getAppModels(app):
30 
31     models = collections.namedtuple('model',['locations','seasons','radars','segments','frames',
       'point_paths','crossovers','layer_groups','layers','layer_links','layer_points','landmarks'])
32 
33     locations = get_model(app,'locations')
34     seasons = get_model(app,'seasons')
35     radars = get_model(app,'radars')
36     segments = get_model(app,'segments')
37     frames = get_model(app,'frames')
38     point_paths = get_model(app,'point_paths')
39     crossovers = get_model(app,'crossovers')
40     layer_groups = get_model(app,'layer_groups')
41     layers = get_model(app,'layers')
42     layer_links = get_model(app,'layer_links')
43     layer_points = get_model(app,'layer_points')
44     landmarks = get_model(app,'landmarks')
45 
46     return models(locations,seasons,radars,segments,frames,point_paths,crossovers,layer_groups,layers,
       layer_links,layer_points,landmarks)
```

Fig. 33: Utility Python functions used by getFrameClosest to parse the input and get the Django models. These functions are used in every OPS Django view.

After *getInput()* is executed the data object must be further parsed into single variables. Lines 33- 35 (Fig. 32) show the parsing of the three required inputs to the function. Optional inputs

are created in the same way except default values are set if the input is not found. Note that all of the logic in the Django views is wrapped in python *try catch* blocks. This allows the server to return an appropriate error in case of failure instead of just failing and never returning a response to the client.

```
59    try:
60
61        if useAllSeasons:
62
63            inSeasonNames = models.seasons.objects.filter(location_id__name=inLocationName,public=True).
          values_list('name',flat=True) # get all the public seasons
64
65        epsg = utility.epsgFromLocation(inLocationName) # get the input epsg
66        inPoint = GEOSGeometry('POINT ('+str(inPointX)+' '+str(inPointY)+')', srid=epsg) # create a point
          geometry object
67
68        # get the frame id of the closest point path
69        closestFrameId = models.point_paths.objects.filter(location_id__name=inLocationName,
          season_id__name__in=inSeasonNames,season_id__public__in=inSeasonStatus,segment_id__name__range=(
          inStartSeg,inStopSeg)).transform(epsg).distance(inPoint).order_by('distance').values_list(
          'frame_id','distance')[0][0]
70
71        # get the frame name,segment id, season name, path, and gps_time from point_paths for the frame
          id above
72        pointPathsObj = models.point_paths.objects.select_related('frames__name','seasons_name').filter(
          frame_id=closestFrameId).transform(epsg).order_by('gps_time').values_list('frame__name',
          'segment_id','season__name','geom','gps_time')
73
74        # get the (x,y,z) coords from pointPathsObj
75        pointPathsGeoms = [(pointPath[3].x,pointPath[3].y,pointPath[4]) for pointPath in pointPathsObj]
76        xCoords,yCoords,gpsTimes = zip(*pointPathsGeoms)
77
78        # build the echogram image urls list
79        outEchograms = utility.buildEchogramList(app,pointPathsObj[0][2],pointPathsObj[0][0])
80
81        # returns the output
82        return utility.response(1,{'season':pointPathsObj[0][2],'segment_id':pointPathsObj[0][1],
          'start_gps_time':min(gpsTimes),'stop_gps_time':max(gpsTimes),'frame':pointPathsObj[0][0],
          'echograms':outEchograms,'X':xCoords,'Y':yCoords,'gps_time':gpsTimes})
83
84    except:
85        return utility.errorCheck(sys)
```

Fig. 34: Logic section of the getFrameClosest Django view.

After the input is parsed (Fig. 32) the actual logic of the function is executed (Fig. 34). Lines 61-63 handle optional input be retrieving a list of all seasons in the database if a list of seasons was not given by the user. Lines 65-66 create a *GEOSGeometry* object from the input X and Y values using the EPSG code (projection code) for the input location. The Django query on line 69 finds the closest frame to the *GEOSGeometry* point object using table relationships in the *point_paths* table. The important part of this query is the chained filter options *.transform(epsg).distance(inPoint).order_by('distance').values_list('frame_id','distance')[0] [0]* which transforms the frame geometries selected, calculates the distance of each frame geometry to the point geometry, orders the results by the distance, creates a list of the outputs and finally returns only the first element (the closest frame) and the *frame_id*. Lines 72-79 perform an additional query to retrieve the geometry and other required output information for the previously found closest frame. Finally line 82 wraps the output in a Python dictionary and uses the utility function *response()* which simply encodes the dictionary to a JSON string and returns an HTTP response object to the web server for return to the client. With the return of data to the client the *getFrameClosest* view is complete. Again, all views follow this pattern and only there internal logic differs.

6.6.6 JavaScript Object Notation

For a client to include data in an HTTP request for use in a view the JSON format is used to encode variables in a structured string object. JSON is a standard notation and has libraries in most programming languages for encoding and decoding. Python, JavaScript, and MATLAB all support the JSON format which covers all of the languages used in the OPS SDI. For example the decoding of input to the getFrameClosest (Fig. 31) view (an X value, Y value, and optionally list of season names) can be examined (Fig. 35). The JSON string is

included in the body of the HTTP request and is extracted by Django (Fig. 33) and then decoded and parsed by Python using either the *json* or *ujson* modules. In this example the JSON string is shown as a hard coded string, normally it would come as an input from the HTTP request. JSON handling in MATLAB and JavaScript will be discussed in sections 6.7.1 and 6.7.2.

```
1  import json
2
3  jsonStr = '{ "properties": { "x": -67020, "y": -1782509,
   "season": [ "2010_Greenland_DC8", "2011_Greenland_P3" ] } }'
4
5  data = json.loads(jsonStr)
6
7  inX = data['properties']['x']
8  inY = data['properties']['y']
9  inSeasons = data['properties']['season']
```

Fig. 35: An example JSON string being decoded and parsed in Python.

6.6.7 URL Dispatcher and WSGI

With the Django Views API defined on the server the Django web framework needs a way to tell the web server how to access each callback function (view). To do this Django leverages a web socket gateway interface (WSGI) to link Apache HTTP and the Django views API. WSGI allows Apache to take a HTTP request in the form of a URL, pass it to Django, wait for a response from Django (generated by each view), and then render that HTTP response back to the client.

```
33        url(r'^get/path$','ops.views.getPath'),
34        url(r'^get/frame/closest$','ops.views.getFrameClosest'),
35        url(r'^get/layers$','ops.views.getLayers'),
36        url(r'^get/layer/points$','ops.views.getLayerPoints'),
37        url(r'^get/layer/points/csv$','ops.views.getLayerPointsCsv'),
```

Fig. 36: A subset of the file urls.py which maps Django views to web URLS using the Django URL Dispatcher.

When Apache HTTP sends an HTTP request to Django the Django URL Dispatcher is used to figure out which view to execute. Fig. 36 shows a subset of the configuration file *urls.py* which maps Django views to specific URLs for the Django URL Dispatcher. Line 34 maps the getFrameClosest view to the URL http://ops.cresis.ku.edu/ops/get/frame/closest. When a client sends a request (with JSON data) to this URL it is passed by Apache HTTP to Django, and then to the getFrameClosest view where it is processed. The view generates an HTTP response and that response is returned by Django to Apache HTTP and then back to the client.

With the URL Dispatcher configured the Django web framework, as used by the OPS is complete. Django offers a much more extensive library of functions not discussed here and a major part of the framework (dynamic templates) is completely unused. Please refer to the Django documentation [42] for more information and the appendices for a link to the complete OPS source code.

6.7 Clients

6.7.1 MATLAB

MATLAB is the primary programming language used by CReSIS for data processing, data management, and creation of data products for distribution therefore the OPS project has developed an API for MATLAB which allows communication between the Django API on the server and any local MATLAB client. The MATLAB API and Data Picker are discussed in the following sections.

6.7.1.1 API

The MATLAB API is a collection of MATLAB scripts that allow communication between the OPS Django API and the MATLAB programming language. The primary role of the MATLAB API is to facilitate communication between the MATLAB Data Picker (section 6.7.1.2) and the OPS but it also allows for the execution of OPS CRUD tasks from MATLAB. The MATLAB API is developed as a separate library and is used as a plugin to the MATLAB *cresis-toolbox*. This means that a directory of organized code (OPS MATLAB API) is stored in another organized directory of code (*cresis-toolbox*) which is added to the MATLAB execution path and used like any other MATLAB functions.

A

```
1  def getFrameClosest(request):
2      """ Gets the closest frame from the OPS database.
3
4      Input:
5          location: (string) region of frame to retrieve
6          x: (float) x value of point to find the closest frame to
7          y: (float) y value of point to find the closest frame to
8
9      Optional Input:
10         season: (string or list of strings) season/s of frame to retrieve
11         startseg: (string) minimum segment name the output frame can belong to
12         stopseg: (string) maximum segment name the output frame can belong to
13         status: (boolean) can be used to get frames for private seasons
14
15     Output:
16         status: (integer) 0:error 1:success 2:warning
17         data:
18             season: (string) the season name of the closest frame
19             segment_id: (integer) the segment id of the closest frame
20             start_gps_time: (float) the start gps time of the closest frame
21             stop_gps_time: (float) the stop gps time of the closest frame
22             frame: (string) the name of the closest frame
23             X: (list of floats) the x coordinates of the closest frame
24             Y: (list of floats) the y coordinates of the closest frame
25             gps_time: (list of floats) the gps times of the closest frame
26             echograms: (list of strings) urls of the ftp echograms of the closest frame
27
28     """
```

B

```
1  function [status,data] = opsGetFrameClosest(sys,param)
2  %
3  % [status,data] = opsGetFrameClosest(sys,param)
4  %
5  % Find the closest frame to a given point from the database.
6  %
7  % Input:
8  %   sys: (string) sys name ('rds','accum','snow',...)
9  %   param: structure with fields
10 %       properties.location = string ('arctic' or 'antarctic')
11 %       properties.x = double
12 %       properties.y = double
13 %       OPTIONAL:
14 %           properties.season = string OR cell of string/s
15 %           properties.startseg = string of minimum segment name
16 %           properties.stopseg = string of maximum segment name
17 %
18 % Output:
19 %   status: integer (0:Error,1:Success,2:Warning)
20 %   data: structure with fields (or error message)
21 %       properties.frame = string
22 %       properties.X = double array
23 %       properties.Y = double array
24 %       properties.gps_time = double array
25 %       properties.frame = string
26 %       properties.season = string
27 %       properties.segment_id = double
28 %       properties.start_gps_time = double
29 %       properties.stop_gps_time = double
30 %
31 % Author: Kyle W. Purdon, Trey Stafford
```

Fig. 37: The Django view (A) and MATLAB API function (B) for getFrameClosest.

Each Django view (Fig. 30) has a corresponding MATLAB function in the MATLAB API. The role of the MATLAB function is to take MATLAB input (a structure), convert it to a JSON string, send it as an HTTP request to the Apache HTTP web server, wait to the HTTP response, and finally decode the response back into a MATLAB structure. Fig. 37 shows the docstrings for the corresponding Django view (A) and MATLAB API function (B) getFrameClosest.

```
1  % CONSTRUCT THE JSON STRUCTURE
2  param.properties.status = [true,false];
3  jsonStruct = struct('properties',param.properties);
4
5  % CONVERT THE JSON STRUCTURE TO A JSON STRING
6  try
7    jsonStr = tojson(jsonStruct);
8  catch ME
9    jsonStr = savejson('',jsonStruct,'FloatFormat','%2.10f');
10 end
11
12 % SEND THE COMMAND TO THE SERVER
13 opsCmd;
14 if gOps.profileCmd
15   [jsonResponse,~] = opsUrlRead(strcat(gOps.serverUrl,'profile'),gOps.dbUser,gOps.dbPswd,...
16     'Post',{'app' sys 'data' jsonStr 'view' 'getFrameClosest'});
17 else
18   [jsonResponse,~] = opsUrlRead(strcat(gOps.serverUrl,'get/frame/closest'),gOps.dbUser,gOps.dbPswd,...
19     'Post',{'app' sys 'data' jsonStr});
20 end
21
22 % DECODE THE SERVER RESPONSE
23 [status,decodedJson] = jsonResponseDecode(jsonResponse);
24
25 % CREATE THE DATA OUPUT STRUCTURE OR MESSAGE
26 data.properties.frame = decodedJson.frame;
27 data.properties.season = decodedJson.season;
28 data.properties.segment_id = double(decodedJson.segment_id);
29 data.properties.start_gps_time = double(decodedJson.start_gps_time);
30 data.properties.stop_gps_time = double(decodedJson.stop_gps_time);
31 data.properties.X = cell2mat(decodedJson.X)';
32 data.properties.Y = cell2mat(decodedJson.Y)';
33 data.properties.gps_time = cell2mat(decodedJson.gps_time)';
34 % data.properties.echograms = decodedJson.echograms;
35
36 end
```

Fig. 38: Body of the MATLAB API function opsGetFrameClosest.

The MATLAB API functions do not perform any of the logic of the task they represent. All of the logic is handled by the Django views on the server. Each MATLAB function simply allows a user to call a MATLAB function using MATLAB variables which are passed to the web server and processed by the corresponding Django view. Fig. 38 shows the MATLAB API function *opsGetFrameClosest* which corresponds to the OPS view

getFrameClosest. Lines 1-10 take a MATLAB structure and convert it to a JSON string for packaging in an HTTP request. A MATLAB MEX [45] function (*tojson*, *fromjson*) based on the cJSON library is used to encode the MATLAB structure to JSON [46]. If for any reason the MEX functions fail a backup library JSONLab [47] (*savejson, loadjson*) is used. The C++ based MEX functions are much faster than the MATLAB native JSONLab library so they are used as the primary conversion tool. Lines 13- 20 generate and submit the HTTP request to the OPS Apache web server and wait for a response. During this time the process discussed in section 6.6 (Django processing) occurs. Once a response is received by MATLAB lines 23-34 decode the HTTP response (and the JSON it contains) and return a MATLAB structure from the MATLAB API function.

```
1   param.properties.x = -67020;
2   param.properties.y = -1782509;
3   param.properties.season = {'2010_Greenland_DC8', '2011_Greenland_P3'};
4
5   jsonStruct = struct('properties',param.properties);
6
7   jsonStr = tojson(jsonStruct);
```

Fig. 39: An example JSON string being encoded in MATLAB.

In section 6.6.6, Fig. 35 showed the basic process of decoding and parsing a JSON string in Python. Fig. 39 shows the same data being encoded into a JSON string by MATLAB. The result variable *jsonStr* will contain the identical string shown as the input variable *jsonStr* in Fig. 35.

6.7.1.2 Data Picker

While the MATLAB API can be used as a standalone library and its functions included in any MATLAB script the primary role of the API is to facilitate the operation of the custom MATLAB Data Picker developed by CReSIS. The Data Picker is a MATLAB tool developed at CReSIS for the manual and automated digitization of ice layers. The Data Picker existed prior the development of the OPS but was built to handle only CReSIS ice surface and ice bottom layers loaded directly from MATLAB layerData files on a local network. The version presented here is a major redesign of the tool which allows the inclusion of any layers and reads and writes OPS server data via the OPS MATLAB API. The Data Picker is one of the two primary clients of the OPS. Although the design of the Data Picker itself is not a direct part of this project, the development of the OPS and the new Data Picker were synchronous and the Data Picker would not function without the OPS MATLAB API and the backend data storage the OPS database provides.

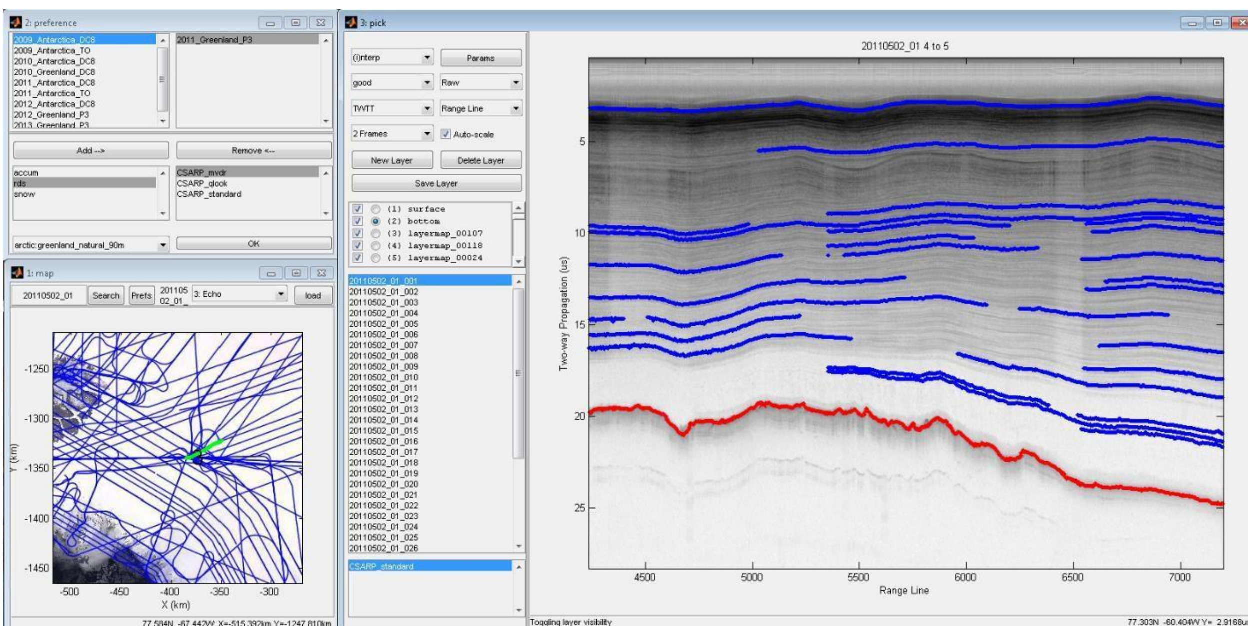

Fig. 40: The CReSIS MATLAB Data Picker user interface.

Fig. 40 shows the user interface (GUI) of the MATLA Data Picker. The GUI consists of three primary windows, the preference window (top left), the map window (bottom left), and the pick window (right). Each of these windows gets data from the OPS server. The preference window is filled with options describing what data is available for viewing in the tool. All of this data comes from the *getSystemInfo* view and includes the available radar systems, seasons, and their respective locations. The preference window also contacts the GeoServer using the MATLAB mapping toolbox to get a list of available WMS images for viewing in the region selected. After making a selection in the preference window the map window is launched. Data in the map window is completely driven by the OPS GeoServer accesed via the WMS request tool in the MATLAB mapping toolbox. The tool is included in MATLAB and not a custom part of the OPS MATLAB API. A line is selected on the map

window by clicking which calls the OPS view *getFrameClosest*. When a selection is made in the map window the pick window is loaded. The radar echogram is loaded from a local file system, but all of the layer points, their layer information, and the list of frames in the segment are pulled from the OPS server using the MATLAB API. Radar echogram data is not stored on the OPS because of its data volume and raster data type. At the current time a solution for storing large volumes of raster data that need to be loaded dynamically (not just as a PNG/JPEG image) has not been identified which offers superior performance to a standard file system. Due to this the CReSIS radar echograms are stored outside of the OPS SDI on the local CReSIS file system. A future task and goal of the OPS development will be to address this issue.

6.7.2 OPS GeoPortal

The OPS GeoPortal is a JavaScript web application that is designed with the discovery and download of geographic content (OPS data) as the primary function. The discovery of geographic content is the primary role of a GeoPortal [48]. The OPS GeoPortal is a complete MVC application developed using the Sencha ExtJS application framework which is an application development platform with cross-browser compatibility, advanced MVC architecture, and a sleek modern UI [49]. ExtJS is distributed freely for web applications that are FOSS which means commercial developers must purchase ExtJS but open source developers can use if free of charge. The expanse suite of tools and methods offered by ExtJS will allow the OPS GeoPortal to continue to develop and change with the needs of the cryosphere community.

The model-view-controller (MVC) application concept was introduced in section 6.6.2 because the Django web framework also follows the MVC pattern. ExtJS follows the same MVC pattern by separating data management, logic, and interface elements within the framework. Data management (M) is handled using ExtJS models and stores. These objects define what type and format background data (models) should be stored as and how and where to retrieve and store the data (stores). The layout (V) or view of the web application is handled using views, which define what user interface (UI) components are shown on the screen and how they are laid out. The logic (C) of the web framework is handled using controllers. Controllers do things like define what happens when you click a button or click on a map. The individual pieces, or components, of an MVC application normally are stored in separate files

in some pre-defined directory structure. ExtJS follows this standard practice and even offers a desktop tool called Sencha Cmd [50] that automatically creates a template directory and contains tools for creating new empty components within the template. Using the Sencha Cmd tool and following the MVC architecture laid out with ExtJS guarantees that the OPS GeoPortal will be a fully MVC compliant web application.

While the ExtJS library contains a vast number of tools and methods it does not contain a native solution for map based content. An additional open source library GeoExt is used to add this functionality to ExtJS [51]. GeoExt is a JavaScript framework which extends the base classes of the ExtJS framework using the OpenLayers (OL) [52] JavaScript mapping library. Because GeoExt extends the existing class structure of ExtJS it blends seamlessly with ExtJS. This means that all of the Sencha Cmd tools and ExtJS functionality operate as normal on the GeoExt extended ExtJS framework. OL is an actively developed and popular open source mapping library. At the time of this thesis the OL community is currently working on a new major release called OpenLayers 3 or OL3 [52]. This release will update OL to use a class like structure similar to ExtJS and GeoEXT and will add new functionality like 3D map interfaces using WebGL (Web Graphics Library). This addition of a 3D based map interface (such as Google Earth) will allow the OPS to eventually simplify its architecture by using a single cylindrical projection displayed on a 3D globe allowing locations like Greenland and Antarctica to be configured using a single map definition instead of two separate maps projected in local polar stereographic projections.

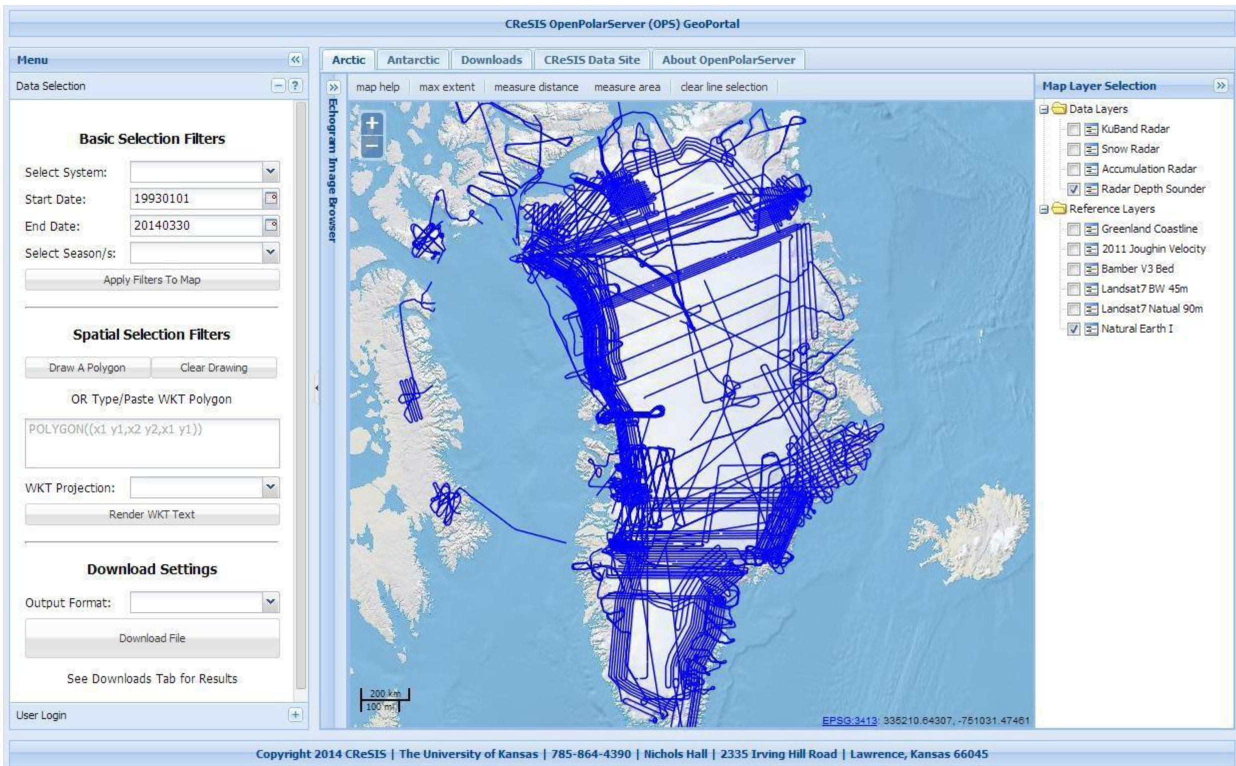

Fig. 41: The OPS GeoPortal user interface.

Combining the functionality of ExtJS, GeoExt, and OpenLayers allowed for the development of the OPS GeoPortal (Fig. 41). The GeoPortal GUI can be broken down into several graphical elements. These elements are separated in the MVC framework meaning that they each have their own view (how the element is shown), model (what data is used by the element), and controller (what the element does). Fig. 42 shows the GeoPortal divided into its two major elements: the *menu panel* and the *tabs panel*. The menu panel contains elements such as the spatial and temporal data filtering options, download output options, and

buttons required to submit various requests. The tabs panel contains a tabbed set of pages (much like tabs in a web browser). Some of the tabs (called *maptabs*) represent the different geographic locations of data (*Arctic, Antarctic*) included in the GeoPortal. Other tabs include an *About* page, the CReSIS FTP data site (shown as an external webpage from within the OPS GeoPortal), and a *Downloads* tab which shows submitted download requests, their status, and a link to completed data for download. Each of the two *maptabs* (*Arctic, Antarctic*) contain additional separate elements including a *layer* selection tree, *map* interface, and *echogram* browsing panel. These are shown in the bottom image of Fig. 42. In ExtJS panels are nested which means the *Arctic:Map* panel is a completely separate element from the *Antarctic:Map* panel. This provides the ability to separate different logic into different files which is the basic idea behind MVC. All of the panels (*Menu, Tabs (Echogram, Map, Layer)*) are collapsible and some are resizable. This means that the actual layout of the GUI can be dynamically adjusted by the user or automatically adjusted by the developer for performing different tasks within the GeoPortal.

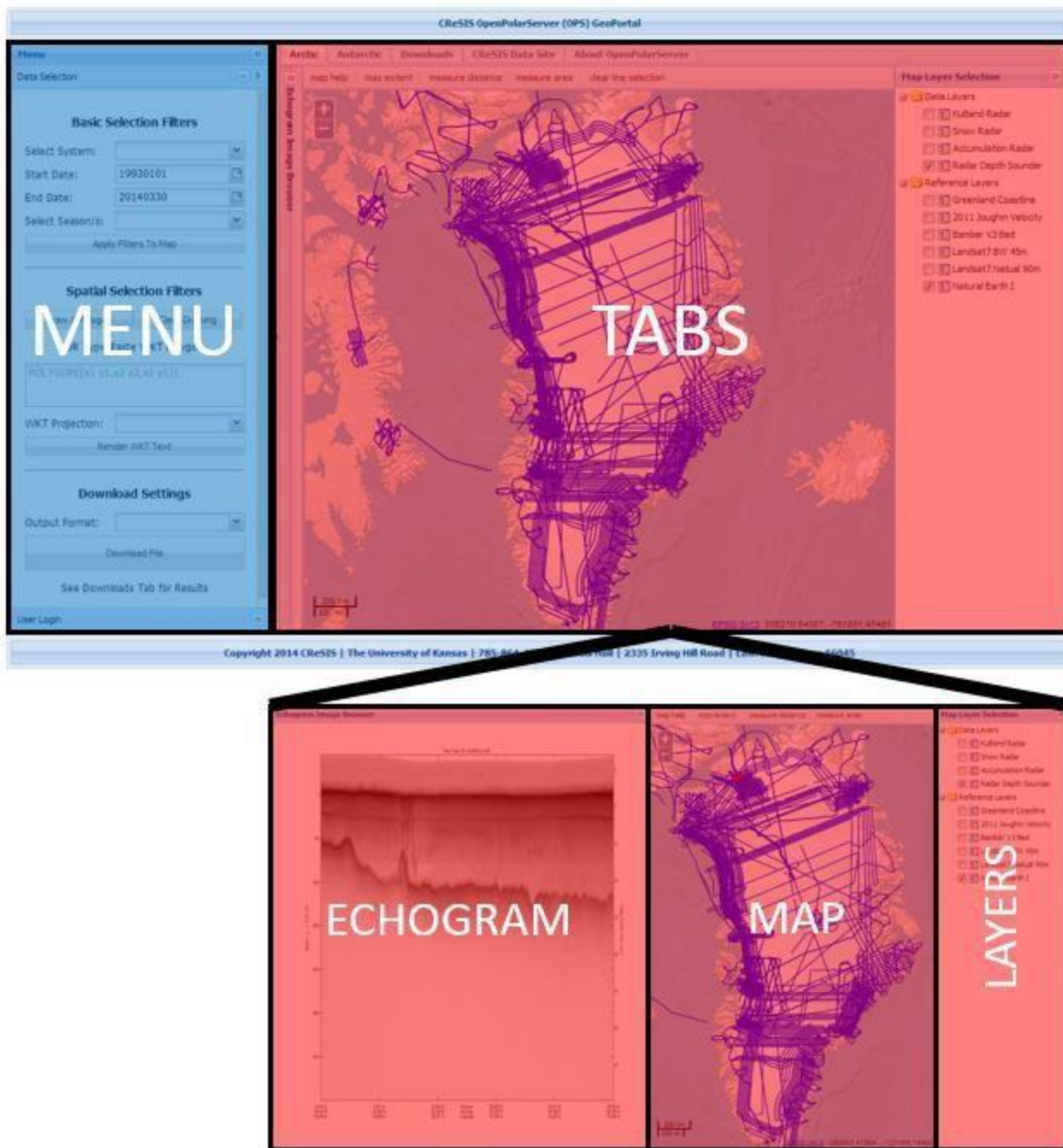

Fig. 42: The OPS GeoPortal GUI divided into its basic elements (top) and a map tab (bottom) divided into its basic elements.

In addition to the discovery and download of OPS data a major feature of the OPS GeoPortal is the ability to spatially browse CReSIS L1B radar echograms. The bottom image in Fig. 42 shows the *Arctic maptab* in echogram browsing mode. This simply means that the *echogram* panel is expanded showing a radar echogram image. The expansion of the *echogram* panel is done automatically when a user starts echogram browsing mode by clicking to select a line in the *map* panel. The *layers* and *menu* panels are also automatically collapsed during this process to allocate the entire browser window to the *map* and *echogram* panels. In its current implementation the radar echograms are just JPEG images loaded and rendered from the CReSIS FTP site. In future development a live echogram browser will be launched that loads the raw CReSIS radar echogram data from the OPS server and allow for dynamic exploration of the image including image processing adjustments, axis adjustments, and a link between the map and image. This dynamic echogram browser will be loosely based on the work done by Christian Panton and his Python based HTML5 radar viewer [53].

While spatially browsing radar echograms is enough to make the GeoPortal useful to the cryosphere community the primary role of the GeoPortal is the discovery and download of data. Users are able to set both temporal (start/stop date, season names) and spatial data filters in the *Menu* panel. Spatial filters are set by allowing the user to either draw a polygon in the *map* panel or allowing them to enter a well-known text (WKT) polygon string and render it to the *map* panel. Once a user sets at least a spatial filter (all temporal filters can be left as defaults) they can select a download format (CSV, CSV-GOOD, KML, or MAT) and submit a request for data. When a user clicks the *Download File* button the *Menu* controller click event for the button is triggered. This event collects all of the possible inputs (start/stop dates, seasons, drawn/rendered boundary …) and encodes them all into a JSON string formatted as

input to the OPS Python Views API. Since JSON (JavaScript Object Notation) is based off of the native JavaScript object a simple call to an object method called *stringify* creates a JSON string from a JSON object. To submit an HTTP request to the Apache web server ExtJS uses the asynchronous JavaScript and XML (AJAX) protocol [54]. The process here is very similar to that of the MATLAB API. The GeoPortal takes user inputs (generated by ExtJS GUI elements) encodes them in a JSON string and sends an HTTP request to the Apache web server calling a corresponding Django view, waits for an HTTP response, and then decodes the response and presents the results to the user. Fig. 43 shows the GUI elements used in the download process. As downloads are completed their status and processing time are rendered to an ExtJS grid panel on the *Downloads* tab. Also included in the *Downloads* grid is a link to the data file on the OPS server ready for download.

Fig. 43: The OPS GeoPortal spatial filter selection (blue) after a polygon has been drawn on the map. Also the download menu (red) and a preview of the Downloads tab populated with completed downloads.

Fig. 44 shows the AJAX request for the download of a CSV file in the GeoPortal. Lines 7-11 define what view will be requested (the URL of the view is given on line 8), and give the input JSON data (line 11). When this function is called AJAX submits an HTTP request and waits for a response in the background. This means that users can still interact (and even submit more downloads) from the GeoPortal GUI. When AJAX receives a response from Apache it enters the *success* block (line 13). This section (lines 22-27) parses the response data (a URL for a file on the server in this case) and writes the results to the

Downloads tab on the GUI. If a known error occurs (the view reports an error with status=0) lines 32-36 are executed which notifies the user of the error via a pop-up alert and writes the error status to the *Downloads* tab.

Line 41 handles any other unknown errors that may occur. The same AJAX format is used to call any of the OPS Python views from the GeoPortal.

```
1   // DEFINE A FUNCTION TO GET A CSV FILE FROM THE OPS
2   function getCsv(selectedSystem,inputJSON,downloadId) {
3
4       // CREATE AN AJAX REQUEST
5       tmp = Ext.Ajax.request({
6
7           method: 'POST',
8           url: '/ops/get/layer/points/csv', // CALL THE getLayerPointsCsv VIEW
9           timeout: 1200000,
10
11          params: {'app':selectedSystem,'data':inputJSON}, //PASS IN THE JSON DATA
12
13          success: function(response){ // ON SUCESS PARSE THE RETURN
14
15              fileDownloadStore = Ext.data.StoreManager.lookup('FileDownloads');
16
17              responseJSON = JSON.parse(response.responseText) // PARSE THE RETURNED JSON
18
19              if (responseJSON.status == 1) { // IF STATUS == 1 (SUCCESS)
20
21                  // WRITE THE STATUS AND DOWNLOAD URL TO THE DOWNLOADS TAB
22                  outRecord = fileDownloadStore.findRecord('id',downloadId);
23                  outRecord.data.status = 'Complete';
24                  outRecord.data.url = responseJSON.data;
25                  outRecord.data.ftime = new Date().toLocaleTimeString();
26                  outRecord.commit()
27                  Ext.Msg.alert('NOTICE','Download is ready, see the Downloads tab.');
28
29              }else {
30
31                  // WRITE THE STATUS (ERROR) TO THE DOWNLOADS TAB
32                  outRecord = fileDownloadStore.findRecord('id',downloadId);
33                  outRecord.data.status = 'Error';
34                  outRecord.data.ftime = new Date().toLocaleTimeString();
35                  outRecord.commit();
36                  Ext.Msg.alert('ERROR',responseJSON.data);
37              }
38          },
39
40          // NOTIFY USER IN AN UNKNOWN ERROR OCCURED
41          failure: function() {Ext.Msg.alert('ERROR','UNKOWN ERROR OCCURED.');}
42      });
43  }
```

Fig. 44: OPS GeoPortal AJAX function for submitting a CSV format download to the OPS Python views API.

The OPS GeoPortal contains thousands of lines of code and therefore cannot be explained in full. To illustrate the basic interaction of MVC components of ExtJS in the GeoPortal the system and season selection menus will be detailed. Systems in the GeoPortal correspond to CReSIS radars: radar depth sounders (rds), accumulation radar (accum), snow radar (snow), and kuband radar (kuband). The *Menu* contains a drop-down box which allows for the selection of a single system. In order to populate this box with the systems stored in the OPS database an ExtJS model and store are used (Fig. 45). A model defines the structure of a data object. In this case the System model is an object with four fields (season, system, location, and public). A store can be defined in a variety of ways, using JSON for local fixed data, or in this case an AJAX request. The Systems store is an AJAX request which automatically sends an HTTP request to the OPS Python view *getSystemInfo* when the GeoPortal loads. This call will create a store which contains a complete list of seasons (and there corresponding system and location) from the OPS database.

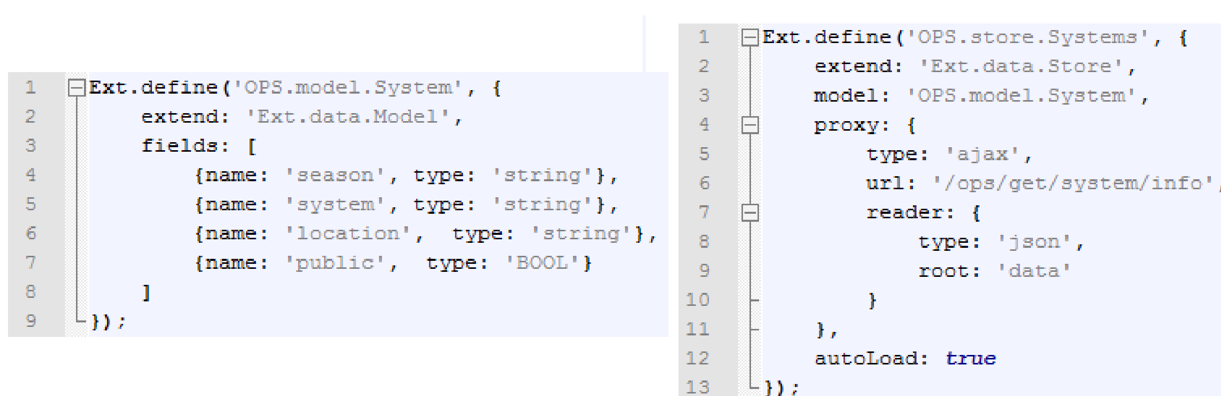

Fig. 45: The OPS GeoPortal ExtJS model (left) and store (right) for the CReSIS radar systems.

When the Systems store is loaded it contains a non-unique list of CReSIS radar systems because there are many seasons with the same radar system. The drop-down menu for systems only needs to present a list of unique systems for the user to select from. In order to do this the ExtJS controller for the *Menu* panel is used. When a user selects the arrow on the systems drop- down menu an event called *focus* is triggered. The *Menu* controller is configured to listen for this event (Fig. 46) and perform some logic when the event is triggered. In order to determine which systems should be displayed in the selection menu the *focus* event function first determines which location (tab) the user is currently viewing (lines 3-4). Next the function retrieves all of the data in the ExtJS Systems store (line 6), clears any filters currently applied to the store (line 7), and uses the *collect* function to get a distinct list of the systems in the store (lines 8-12). Finally the function gets the drop-down menu object (line 14), creates a new store of the distinct systems (lines 16-19) and binds the store to the drop-down menu object (line 21) exposing the list of distinct systems to the user for selection in the menu.

```
change: function() {

    var curSystem = Ext.ComponentQuery.query('#selectedSystem')[0].value

    var systemStore = Ext.getStore('Systems');
    systemStore.clearFilter()
    systemStore.filter('system',curSystem);
    systemStore.filter('location',curLocation);
    systemStore.filter('public',true);
    var distinctSeasons = systemStore.collect('season');
    var outSeasons = [];
    for (var i=0;i<distinctSeasons.length;i++){
        outSeasons.push([distinctSeasons[i]]);
    }
    var seasonCombo = Ext.ComponentQuery.query('#selectedSeasons')[0]

    distinctSeasonsStore = new Ext.data.ArrayStore({
        fields: ['season'],
        data: outSeasons
    });

    distinctSeasonsStore.sort('season','ASC');

    seasonCombo.bindStore(distinctSeasonsStore);
}
```

Fig. 46: The ExtJS controller listener for the change event of the systems drop-down menu.

Each time a user selects a new system from the drop-down menu another event called *change* is triggered. The GeoPortal uses this event to update another drop-down menu which lists the CReSIS data seasons available for the selected radar system. Again the *Menu* controller is configured to listen for this event (Fig. 47) and perform some logic when the event is triggered. First the *change* event function gets the system currently selected by the user (line 3). The function next loads the Systems store (line 5), clears any filters currently applied to the store (line 6), and then uses the *filter* function to get a store which only contains seasons for the selected system, location, and that have a True value for the public field (lines

7-9) and creates a list of the distinct seasons (lines 10-14). Finally the function gets the drop-down menu object (line 15), creates a new store of the distinct seasons (lines 17-20) and binds the store to the drop- down menu object (line 22) exposing the list of distinct seasons for the selected system to the user for selection in the menu.

```
1  change: function() {
2
3      var curSystem = Ext.ComponentQuery.query('#selectedSystem')[0].value
4
5      var systemStore = Ext.getStore('Systems');
6      systemStore.clearFilter()
7      systemStore.filter('system',curSystem);
8      systemStore.filter('location',curLocation);
9      systemStore.filter('public',true);
10     var distinctSeasons = systemStore.collect('season');
11     var outSeasons = [];
12     for (var i=0;i<distinctSeasons.length;i++){
13         outSeasons.push([distinctSeasons[i]]);
14     }
15     var seasonCombo = Ext.ComponentQuery.query('#selectedSeasons')[0]
16
17     distinctSeasonsStore = new Ext.data.ArrayStore({
18         fields: ['season'],
19         data: outSeasons
20     });
21
22     distinctSeasonsStore.sort('season','ASC');
23
24     seasonCombo.bindStore(distinctSeasonsStore);
25 }
```

Fig. 47: The ExtJS controller listener for the focus event of the systems drop-down menu.

While AJAX is used by ExtJS to access the OPS Python API using the Django web framework the map images displayed in the GeoPortal must be generated by the OPS GeoServer. The OpenLayers JavaScript library handles this using built in WMS request object,

similar to how MATLAB uses the mapping toolbox to request GeoServer images in the Data Picker. OpenLayers handles all of the map interaction of the GeoPortal including navigation (zoom/pan), map click events, and drawing features on the map.

A useful utility feature of the GeoPortal is the ability to measure distances and areas with OpenLayers measuring tools. The OpenLayers library comes with built-in measuring tools but for them to show live distances/areas as measurements are being made requires the development of additional code. An external OL class plugin developed by Xavier Mamano, OL- DynamicMeasure [55], already existed and offered many features non-existent in the OL default library. This plugin was chosen to avoid replication of already existing work, following the principle of DRY (don’t repeat yourself), or more fitting DRO (don’t repeat others). Fig. 48 shows completed distance and area measurements on the map.

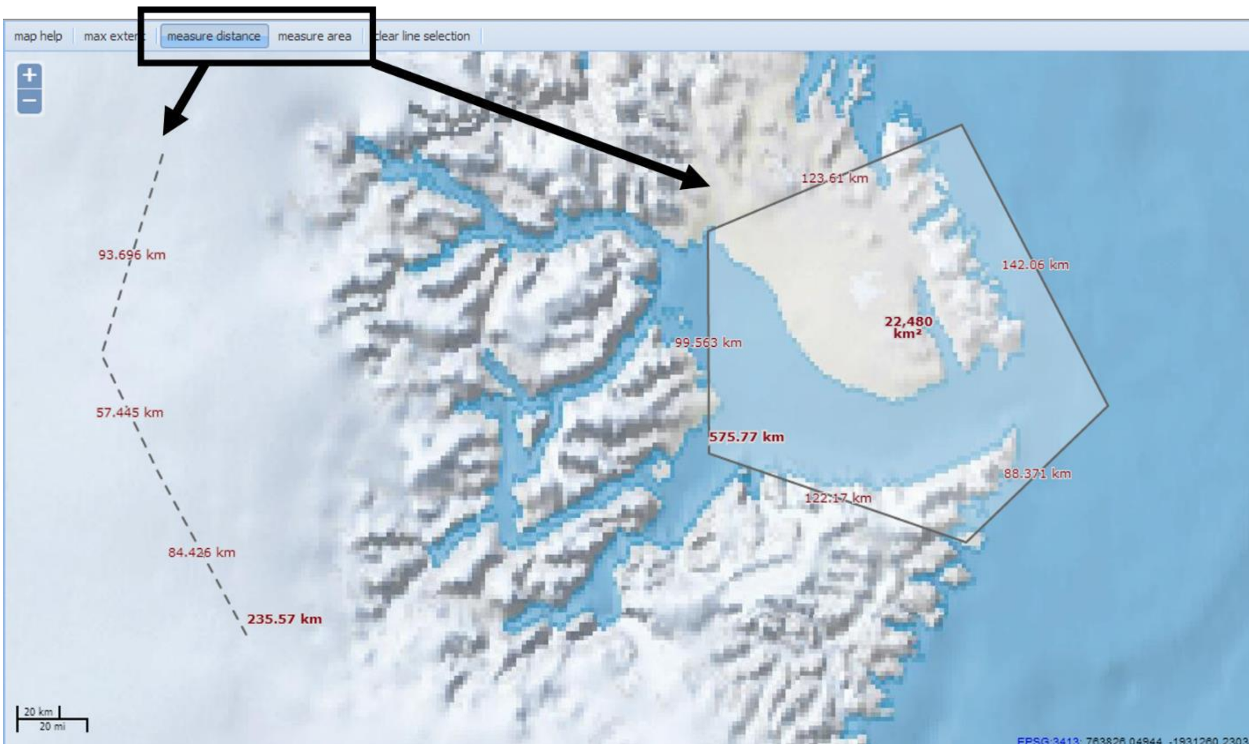

Fig. 48: Distance and Area measuring tools available in the OPS GeoPortal.

While there is much more development required to build the OPS GeoPortal the above examples provides some insight into how the MVC architecture allows a user to perform the task of downloading data, browsing echograms, and measuring features. The OPS GeoPortal is the face of the OPS and therefore is extremely important to its success and acceptance in the cryosphere community. Developing the portal using open source software and following relevant best practices and standards for web application development ensures the system can easily be upgraded as technology advances and new needs within the cryosphere community are identified.

6.8 Vagrant and VirtualBox

One of the requirements of the OPS SDI is that users be able to deploy the complete SDI on laptops for use during remote fieldwork. Without some additional development this means a developer would need to install and configure the entire SDI manually each time a user wanted to deploy the OPS in the field. This, for obvious reasons, is not feasible. The solution was to develop the OPS using a provisioning software called Vagrant [56] and create the complete SDI inside of a virtual machine [57] using Oracle VirtualBox [58]. A virtual machine (VM) is a software implementation of a computer. A VM allows a single set of physical hardware (Hard Drive, Random Access Memory, Processors) to be shared by many separate VMs each allocated a certain portion of the real hardware. Installing all of the OPS SDI software (Apache HTTP, Apache Tomcat, Django, etc…) inside of a VM on a field user's computer ensures that there will be no installation conflicts with software installed on the computers own operating system. The VM alone does not solve all the issues, without Vagrant, a developer would still need to manually install the VM and all of the OPS SDI software. Vagrant allows the definition of the installation of an entire operating system and any software in two simple text files, a *Vagrantfile* (Fig. 49) and *Provisions* file (Fig. 50).

```
1   # -*- mode: ruby -*-
2   # vi: set ft=ruby :
3
4   # Vagrantfile API/syntax version. Don't touch unless you know what you're doing!
5   VAGRANTFILE_API_VERSION = "2"
6
7   Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
8
9       # VAGRANT BASE BOX
10      config.vm.box = "CentOS65"
11      config.vm.box_url = "https://dl.dropboxusercontent.com/u/101242742/v2/CentOS65.box"
12
13      # FORWARDED PORT MAPPING
14      config.vm.network :forwarded_port, guest: 80, host: 80
15
16      # SET UP STATIC PRIVATE IP
17      config.vm.network :private_network, ip: "192.168.111.222"
18
19      # VIRTUALBOX SPECIFIC CONFIGURATION
20      config.vm.provider :virtualbox do |vb|
21
22          # BOOT INTO A GUI
23          vb.gui = true
24
25          # CHANGE THE NAME
26          vb.name = "OpenPolarServer"
27
28          # CHANGE THE HARDWARE ALLOTMENT
29          vb.customize ["modifyvm", :id, "--memory", "1024"]
30          vb.customize ["modifyvm", :id, "--cpus", "1"]
31
32      end
33
34      # ENABLE SHELL PROVISIONING
35      config.vm.provision :shell, :path => "conf/provisions.sh"
36
37  end
```

Fig. 49: The OPS SDI Vagrantfile used by Vagrant to create the OPS VM.

The Vagrantfile (Fig. 49) tells the Vagrant software which base box (a pre-configured operating system with no software installed) to use (lines 10-11), what network ports and IP to use (lines 14-17) and configures the VM by setting the VM name and hardware allocations (lines 20-32). Finally it defines a *Provisions* file which will be executed when the VM starts (line 35).

The Vagrant *Provisions* file (Fig. 50) is a Linux shell script which contains all of the Linux terminal commands needed to install and configure all of the OPS SDI software. The provisions file is too long to detail completely but the installation of Python 2.7.6 in a virtual environment (lines 121-135) and Apache HTTP and WSGI (141-153) are shown in Fig. 50. The provisions file takes a fresh CentOS Linus OS (defined by the vagrant box in the *Vagrantfile*) and installs all software, custom files, and configurations resulting in a complete and fully functioning OPS SDI. All of the software discussed in sections 6.2 through 6.6 are installed during this provisioning process.

```
117 # ----------------------------------------------------------------------
118 # INSTALL PYTHON 2.7 AND VIRTUALENV WITH DEPENDENCIES
119 
120 # INSTALL DEPENDENCIES
121 yum groupinstall -y "Development tools"
122 yum install -y wget python-pip zlib-devel bzip2-devel openssl-devel ncurses-devel sqlite-devel readline-devel tk-devel
123 python-pip install --upgrade nose
124 
125 # DOWNLOAD AND INSTALL PYTHON 2.7.6
126 wget http://www.python.org/ftp/python/2.7.6/Python-2.7.6.tar.xz
127 tar xf Python-2.7.6.tar.xz
128 cd Python-2.7.6
129 ./configure --prefix=/usr --enable-shared LDFLAGS="-Wl,-rpath /usr/lib"
130 make && make altinstall
131 
132 # INSTALL AND ACTIVATE VIRTUALENV
133 pip install virtualenv
134 virtualenv -p /usr/bin/python2.7 /usr/bin/venv
135 source /usr/bin/venv/bin/activate
136 
137 # ----------------------------------------------------------------------
138 # INSTALL APACHE WEB SERVER AND MOD_WSGI
139 
140 # INSTALL APACHE HTTPD
141 yum install -y httpd httpd-devel
142 
143 # DOWNLOAD AND INSTALL MOD_WSGI (COMPILE WITH Python27)
144 cd ~ && wget https://modwsgi.googlecode.com/files/mod_wsgi-3.4.tar.gz
145 tar xvfz mod_wsgi-3.4.tar.gz
146 cd mod_wsgi-3.4/
147 ./configure --with-python=/usr/bin/python2.7
148 LD_RUN_PATH=/usr/lib make && make install
149 rm -f ~/mod_wsgi-3.4.tar.gz
150 rm -f ~/mod_wsgi-3.4
151 
152 # INCLUDE THE SITE CONFIGURATION FOR HTTPD
153 echo "Include /var/www/sites/"$serverName"/conf/"$appName".conf" >> /etc/httpd/conf/httpd.conf
```

Fig. 50: A subset of the Vagrant provisions file which shows the installation of Python and Apache HTTP.

While the CReSIS hosted VM is set up once and hosted on a high availability (HA) server at CReSIS any OPS users who wish to run their own complete local copy of the OPS SDI can do so by installing Vagrant, VirtualBox, and downloading the OPS GitHub repository [59]. They can then simple run the command *vagrant up* from the download OPS directory on a command line and the complete OPS SDI will be created and run locally on their own VM.

7 Usability Analysis

In order to verify that the OPS has in fact achieved the goal of providing an improved experience for end users of polar remote sensing data a numerical comparison of the usability of each one of the four existing systems was completed. In the comparison, a single user completed the same task, “download CReSIS L2 data in the csv-good format over a NE Greenland outlet glacier (79N) including only data from 2010 and 2011”, on each system.

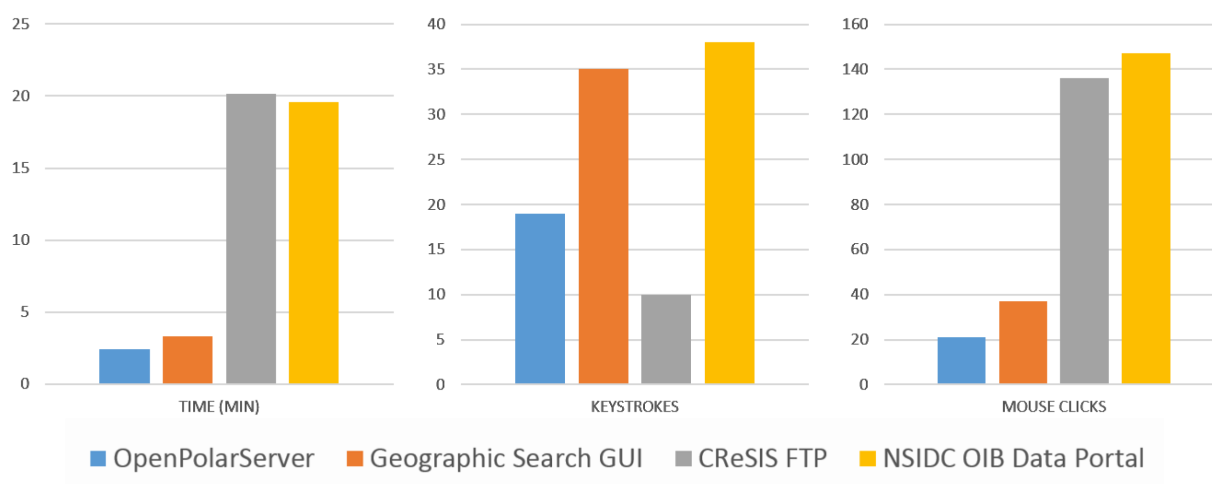

Fig. 51: Results from a usability test of four systems. The processing time, number of keystrokes, and number of mouse clicks required to download the same dataset are shown for each system.

The usability results show that the OPS has achieved the goal of providing an improved experience. Fig. 51 shows that the OPS had the fastest processing time, the smalles number of mouse clicks, and the second least number of keystrokes. Mouse clicks and keystrokes indicate the simplicity of the user interface of a system. A system which

requires the lowest number of each generally is simpler to use. Note that while the CReSIS FTP recorded the least number of keystrokes they were likely offset by mouse clicks which is the second highest number in that category. In fact when the number of clicks and keystrokes are combined the OPS recorded the lowest total indicating the OPS system is simpler to use than the other systems.

It should be noted, however, that these results are from a single user performing one task and may not necessary represent an average situation over a set of users and tasks. These results also present the ideal case (i.e., perfect use of each system where end users are already familiar with the systems through tutorials and help documentation). The OPS does include interactive help documentation that should make it very easy for new users to familiarize themselves with the system quickly.

8 Conclusions

In section 4.3 the tasks and goals of the OPS SDI project were outlined. This thesis presented the methods used to complete each of these goals. PostgreSQL, PostGIS, and Django were used to develop and deploy a database management system. ExtJS, GeoExt, and OpenLayers were used to develop and deploy a Geoportal. Custom MATLAB scripts and the Django Python web framework were used to deploy an API for the interaction between MATLAB and the OPS. The reduced effort (as shown by the usability analysis in chapter 7) needed to acquire and analyze data will allow and encourage new scientists to explore the data and potentially provide new scientific knowledge in an effort to understand the cryosphere, ice sheets, and future sea level rise. The achievement of these goals marks the completion of the development and distribution of the OPS SDI. It is the author's hope that the cryosphere community accepts the system, uses it, and contributes to its future. During the development of the OPS many lessons were learned and there are still many to learn. Software development is an ever changing field and it is very likely that new developments in the near future will better serve various features of the OPS SDI. It is for this reason that continuous maintenance and upgrades to the project continue.

The OPS has been developed using all FOSS and conforms to practical data and coding standards. The system is public at https://www.ops.cresis.ku.edu and many cryosphere community data providers have or are preparing to include their datasets in the OPS. The code is open source and provided free of charge or restriction on GitHub at https://github.com/CReSIS. As an open source project the OPS will only be successful if the community contributes to its growth. Some possibilities for future exploration of the OPS are:

1. Cloud-based hosting of the SDI on systems such as RACKSPACE [60] or Amazon EC2 [61].
2. An interactive (web based) data picking system build in JavaScript to replace the MATLAB data picker.
3. An interactive JavaScript echogram browser that loads dynamic data instead of static images.
4. A web based data loading system so community members can load their own data into a centrally hosted system without CReSIS intervention.

In addition to those future development opportunities the maintenance and continuous software upgrade of the OPS is also a task that must be accounted for. CReSIS IT staff, with the aid of future graduate students, will continue to maintain and develop the system over its lifetime. The effort and care taken during the development, i.e., following data and coding standards as well as creating comprehensive technical documentation, will simplify future development and maintenance of the OPS.

9 References

[1] CReSIS, “About | Center for Remote Sensing of Ice Sheets.” [Online]. Available: https://www.cresis.ku.edu/about. [Accessed: 31-Oct-2013].

[2] R. Epperson, “Personal Communication,” Lawrence, Kansas USA, 2013. [3] P. Gogineni,

“CReSIS Data.” Lawrence, Kansas USA, 2012.

[4] National Aeronotics and Space Administrasion (NASA), “Data Processing Levels.” [Online]. Available: http://science.nasa.gov/earth-science/earth-science-data/data-processing-levels-for-eosdis-data-products/. [Accessed: 31-Oct-2013].

[5] J. A. MacGregor, M. A. Fahnestock, G. A. Catania, J. D. Paden, S. Gogineni, S. C. Rybarski, S. K. Young, A. . Mabrey, and B. . Wagman, “Radiostratigraphy of the Greenland Ice Sheet,” *manusript in preperation*.

[6] W. B. Krabill, “IceBridge ATM L2 Icessn Elevation, Slope, and Roughness.” National Snow and Ice Data Center, Boulder, Colorado USA.

[7] B. Blair and M. Hofton, “IceBridge LVIS L2 Geolocated Surface Elevation Product.” National Snow and Ice Data Center, Boulder, Colorado USA.

[8] T. Haran, T. Bohlander, T. Scambos, T. Painter, and M. A. Fahnestock, “MODIS mosaic of Antarctica (MOA) image map.” National Snow and Ice Data Center, Boulder, Colorado USA, 2005.

[9] National Aeronotics and Space Administrasion (NASA), “Landsat ETM+ Arctic Mosaic.” Global Mapper, 2013.

[10] I. Joughin, B. Smith, I. Howat, and T. Scambos, “MEaSUREs Greenland Ice Velocity Map from InSAR Data.” National Snow and Ice Data Center, Boulder, Colorado, USA, 2010.

[11] P. Fretwell, H. D. Pritchard, D. G. Vaughan, J. L. Bamber, N. E. Barrand, R. Bell, C. Bianchi, R. G. Bingham, D. D. Blankenship, G. Casassa, G. Catania, D. Callens, H. Conway, a. J. Cook, H. F. J. Corr, D. Damaske, V. Damm, F. Ferraccioli, R. Forsberg, S. Fujita, Y. Gim, P. Gogineni, J. a. Griggs, R. C. a. Hindmarsh, P. Holmlund, J. W. Holt, R. W. Jacobel, a. Jenkins, W. Jokat, T. Jordan, E. C. King, J. Kohler, W. Krabill, M. Riger-Kusk, K. a. Langley, G. Leitchenkov, C. Leuschen, B. P. Luyendyk, K. Matsuoka, J. Mouginot, F. O. Nitsche, Y. Nogi, O. a. Nost, S. V. Popov, E. Rignot, D. M. Rippin, a. Rivera, J. Roberts, N. Ross, M. J. Siegert, a. M. Smith, D. Steinhage, M. Studinger, B. Sun, B. K. Tinto, B. C. Welch, D. Wilson, D. a. Young, C. Xiangbin, and a. Zirizzotti, “Bedmap2: improved ice bed, surface and thickness datasets for Antarctica,” *Cryosph.*, vol. 7, no. 1, pp. 375–393, Feb. 2013.

[12] J. Bamber, "Greenland 5 km DEM, Ice Thickness, and Bedrock Elevation Grids." National Snow and Ice Data Center, Boulder, Colorado USA, 2001.

[13] "Natural Earth Data," 2013. [Online]. Available: http://www.naturalearthdata.com/. [14]

Mathworks, "Matfile_format," 2013. [Online]. Available: http://www.mathworks.com/help/pdf_doc/matlab/matfile_format.pdf. [Accessed: 31-Oct-2013].

[15] "Center for Remote Sensing of Ice Sheets FTP." [Online]. Available: ftp://ftp.cresis.ku.edu/data/.

[16] "National Snow and Ice Data Center." [Online]. Available: http://nsidc.org/.

[17] "Operation Ice Bridge Data Portal." [Online]. Available: http://nsidc.org/icebridge/portal/.

[18] "PolarGrid Cloud GIS." [Online]. Available: http://polargrid.org/polargrid/software-release.

[19] Z. Guo, R. Singh, and M. Pierce, "Building the PolarGrid portal using web 2.0 and OpenSocial," *Proc. 5th Grid Comput. Environ. Work. - GCE '09*, p. 1, 2009.

[20] K. W. Purdon, "CReSIS Level-III Workflow Creation and Data Standardization," in *University of Kansas Undergraduate Research Symposium*, 2011.

[21] B. Clinton, "Coordinating Geographic Data Acquisition and Access : The National Spatial Data Infrastructure," *Fed. Regist. Pres. Doc.*, vol. 59, no. 71, p. 4, 1994.

[22] A. Rajabifard and I. Williamson, "Spatial data infrastructures: concept, SDI hierarchy and future directions," *Proc. GEOMATICS'80 …*, 2001.

[23] S. Steiniger and A. J. S. Hunter, "The 2012 free and open source GIS software map – A guide to facilitate research, development, and adoption," *Comput. Environ. Urban Syst.*, vol. 39, pp. 136–150, May 2013.

[24] S. Steiniger and A. J. S. Hunter, "Free and Open Source GIS Software for Building a Spatial Data Infrastructure," in *Geospatial Free and Open Source Software in the 21st Century*, E. Bocher and M. Neteler, Eds. Springer Berlin Heidelberg, 2012, pp. 247–261.

[25] Wikipedia, "Database," *Wikipedia*. [Online]. Available: http://en.wikipedia.org/wiki/Database.

[26] Open Geospatial Consortium Inc., "OpenGIS ® Web Map Server Implementation Standard," 2006. [Online]. Available: http://www.opengeospatial.org/standards/wms. [Accessed: 31-Oct-2013].

[27] Open Geospatial Consortium Inc., “OpenGIS ® Web Feature Service Implementation Standard,” 2010. [Online]. Available: http://www.opengeospatial.org/standards/wfs.

[28] Open Geospatial Consortium Inc., “OpenGIS ® Web Map Tile Service Implementation Standard,” 2010. .

[29] Open Geospatial Consortium Inc., “Open Geospatial Consortium Inc.,” 2013. [Online]. Available: http://www.opengeospatial.org/. [Accessed: 07-Nov-2013]. [30] “About CentOS.” [Online]. Available: http://www.centos.org/about/. [31] RedHat, “RedHat Enterprise Linux.”

[32] S. Vidal, “YUM Package Manager.”

[33] RedHat, “RedHat Package Manager (RPM).”

[34] “About PostgreSQL.” [Online]. Available: http://www.postgresql.org/about/. [35] “PostGIS.” [Online]. Available: http://postgis.net/.

[36] Wikipedia, “Datbase Design.” [Online]. Available: http://en.wikipedia.org/wiki/Database_design.

[37] “The Apache Software Foundation.” [Online]. Available: http://www.apache.org/. [38] OpenLayers, “OpenLayers ProxyHost CGI.”

[39] “GeoServer.” [Online]. Available: http://geoserver.org/display/GEOS/Welcome. [40] GeoServer, “GeoServer Layer Preview.”

[41] “GeoServer Documentation.” [Online]. Available: http://docs.geoserver.org/stable/en/user/.

[42] “Django Web Framework.” [Online]. Available: https://www.djangoproject.com/. [43] Wikipedia, “Model-View-Controller (MVC).”

[44] “Django MVC FAQ.” [Online]. Available: https://docs.djangoproject.com/en/1.6/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names.

[45] MATLAB, “Introducing Mex Files.”

[46] D. Gamble, “cJSON library.” [Online]. Available: http://cjson.sourceforge.net/.

[47] Q. Fang, “JSONlab toolbox.” [Online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/33381-jsonlab-a-toolbox-to-encodedecode-json-files-in-matlaboctave.

[48] M. G. Tait, “Implementing geoportals: applications of distributed GIS,” *Comput. Environ.Urban Syst.*, vol. 29, no. 1, pp. 33–47, Jan. 2005.

[49] Sencha, “Ext Js.” [Online]. Available: http://www.sencha.com/products/extjs/.

[50] Sencha, “Sencha Cmd.” [Online]. Available: http://www.sencha.com/products/sencha-cmd/download.

[51] “GeoExt 2.” [Online]. Available: http://geoext.github.io/geoext2/.

[52] OpenLayers, “OpenLayers 3.” [Online]. Available: http://ol3js.org/.

[53] C. Panton, “Radar Viewer.”

[54] Wikipedia, “AJAX.” [Online]. Available http://en.wikipedia.org/wiki/Ajax_(programmig).

[55] X. Mamano, “OL-DynamicMeasure,” *GitHub*. [Online]. Available: https://github.com/jorix/OL-DynamicMeasure.

[56] HashiCorp, “About Vagrant.” [Online]. Available:

http://www.vagrantup.com/about.html. [57] Wikipedia, “Virtual Machine.” [Online]. Available: http://en.wikipedia.org/wiki/Virtual_machine.

[58] Oracle, “VirtualBox.” [Online]. Available: https://www.virtualbox.org/.

[59] K. W. Purdon, T. Stafford, and J. D. Paden, “OpenPolarServer GitHub Repository.” .

[60] OpenStack, “RackSpace Cloud Servers.” [Online]. Available: http://www.rackspace.com/cloud/servers/.

[61] Amazon, “Amazon EC2.”

10 Appendices

10.1 Source Code

The OPS source code is available for free (both in cost and application) on GitHub at https://github.com/CReSIS. There are multiple projects on GitHub described in the following table.

Project Name	Description	Link
OPS	The primary source code repository.	https://github.com/CReSIS/OPS
OPS-GEOPORTAL	The complete geoportal source code. Only a compiled version is included in the OPS repository.	https://github.com/CReSIS/OPS-GEOPORTAL
OPS-MATLAB	The OPS MATLAB API source code.	https://github.com/CReSIS/OPS-MATLAB

10.2 Web Application Links

The web applications used for pre-OPS data access can be viewed/used at the following URL's.

Project Name	Description	Link
Geographic Search GUI	MATLAB subsetting tool for the CReSIS FTP.	ftp://data.cresis.ku.edu/data/geographic_search
PolarGrid Cloud GIS	Non-Active reference page for the IU PolarGrid cloud GIS.	http://www.polargrid.org/polargrid/cloud-gis-service
NSIDC OIB Data Portal	The operation NSIDC Operation Ice Bridge GeoPortal.	http://nsidc.org/icebridge/portal/