SWHV CCN3
Design

SWHV-ROB-TN-001-DDF3 v1.1

SWHV Team

Contents

List of Tables

List of Figures

```
id: 1e316562be40ab7db808b78787332cd99a69ccd9
```

1 Introduction

id: 1e316562be40ab7db808b78787332cd99a69ccd9

Table 1.1: Document history

Date	Notes
2020-06-10	Version 0.9 (based on SWHV-DDF2 1.5, CCN3 CDR pre-assessment)
2020-09-17	Version 1.0 (post-CCN3 CDR): • Link CCN3 tasks with WPs.
2021-11-26	Version 1.1 (CCN3 QR): • Update Chapter 6 with work performed up to QR.

Table 1.2: Distribution list

Entity	Team/Role	Representative
ROB	SWHV Team	Bogdan Nicula
FHNW	SWHV Team	André Csillaghy
ESAC	Technical Officer	Pedro Osuna
ESTEC	Technical Officer	Daniel Müller

1.1 Purpose and scope

This document (SWHV-DDF3) is the design study report of the work performed during the CCN3 phase of "Space Weather Helioviewer" project (Contract No. 4000107325/12/NL/AK, "High Performance Distributed Solar Imaging and Processing System" ESTEC/ITT AO/1-7186/12/NL/GLC). It focuses on the detailed explanation of the design for several software components.

1.2 Applicable documents

[1] Contract Change Notice No. 2: SWHV-CCN2-Proposal3-BN2.pdf

[2] Contract Change Notice No. 3: SWHV_CCN3_Proposal_v3.1.pdf

1.3 Reference documents

[1] SWHV CCN3 System Requirements: SWHV-ROB-RS-001-SRD3-v1.1.pdf

2 Work logic

In the following JHelioviewer and SWHV are used interchangeably. They refer to the Java client of the Helioviewer system available at https://github.com/Helioviewer-Project/JHelioviewer-SWHV and subject of this project.

The current system architecture is presented in Chapter 3, the interfaces of the JHelioviewer client are presented in Chapter 4, the current design of JHelioviewer is presented in Chapter 5, while the Chapter 6 presents the identified tasks for the CCN3 phase.

Chapter 7 presents a traceability matrix for the requirements, as well as their assigned priority and the status of their implementation. Requirements already implemented may still further be subjected to refinement and refactoring as new functionality becomes available in the client-server system.

3 System architecture

The following figure depicts the architecture of the Helioviewer system as installed on the ROB server. For the purpose of this project, the focus is on the interaction between the JHelioviewer client and the Helioviewer server.

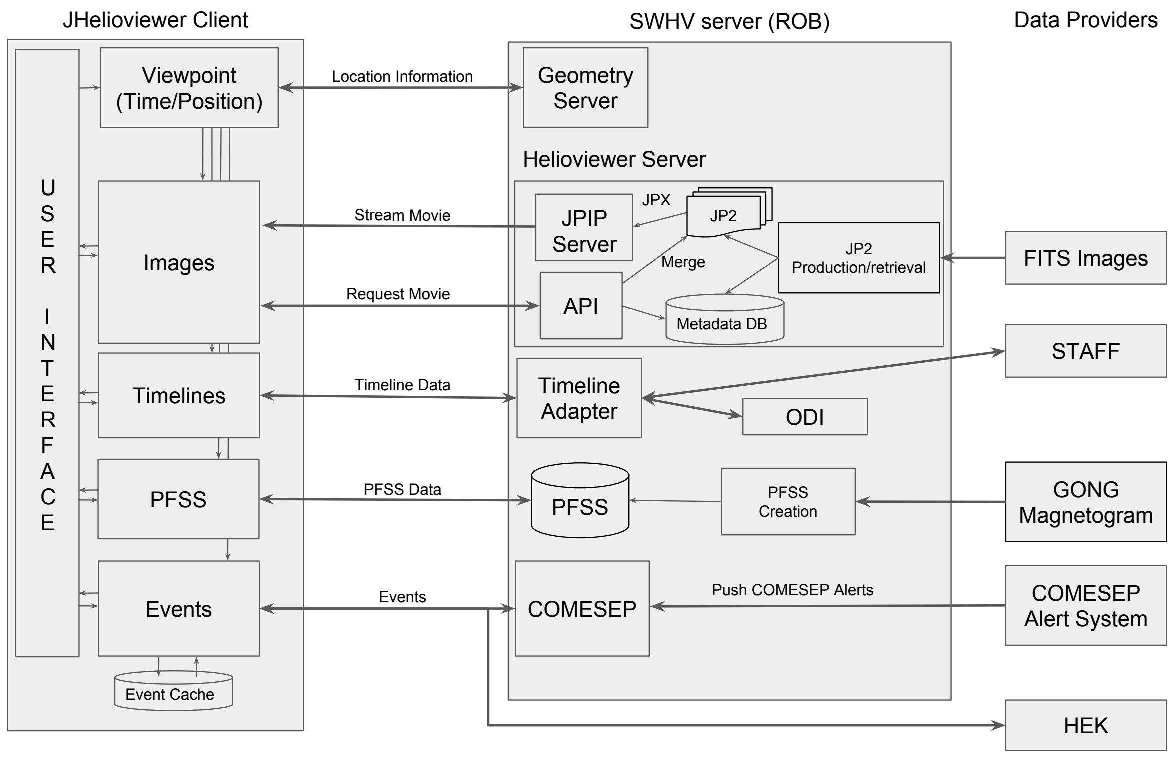

Figure 3.1: Helioviewer system architecture

3.1 Server infrastructure

The following servers are included:

- HTTP server (e.g., Apache or nginx) to serve static files, to proxy HTTP requests, and to run various services:
  - Image services API[1]: It lists the available image datasets and commands the creation of JPX

[1] https://github.com/Helioviewer-Project/api

  movies on demand. It includes a facility to ingest new images files. Metadata about the image files is stored in a MySQL database.
  - Timeline services API: This is an adapter brokering between the JHelioviewer client and the backend timeline storage services (ODI – https://spitfire.estec.esa.int/trac/ODI/ and STAFF backend – http://www.staff.oma.be). It lists the available timeline datasets and serves the data in a JSON format.
  - A PFSS dataset, static FITS files produced regularly out of GONG magnetograms. The JHelioviewer client retrieves them on demand, based on monthly listings (e.g., http://swhv.oma.be/magtest/pfss/2018/01/list.txt).
  - COMESEP service which subscribes to the COMESEP alert system (not part of this project), stores the alerts and makes them available to the JHelioviewer server in a JSON format.
  - The Helioviewer web client[2], not relevant for this project.

- The `esajpip`[3] server, which delivers the JPEG2000 data streams to the JHelioviewer client using the JPIP protocol, built on top of the HTTP network protocol.
- The `GeometryService`[4] server implements a set of high precision celestial computation services based on NASA's Navigation and Ancillary Information Facility (NAIF) SPICE Toolkit and communicates with the JHelioviewer client using a JSON format.
- The `PropagationService`[5] server is currently a mock-up server with the aim in aiding the correlation of in-situ data with remote sensing data.
- The HEK server (maintained by LMSAL at https://www.lmsal.com/hek/, not part of this project) which serves JSON formatted heliophysics events out of HER. JHelioviewer retrieves a curated list of space weather focused events.

To ensure encapsulation, reproducibility, and full configuration control, the services which are part of this project are currently being containerized at https://gitlab.com/SWHV/SWHV-COMBINED.

3.2 JPEG2000 infrastructure

3.2.1 JPIP server

The `esajpip` server serves the JPEG2000 encoded data to the JHelioviewer client. This software was forked from the code at https://launchpad.net/esajpip. It was ported to a CMake build system and to C++11 standard features. Several bugs, vulnerabilities, and resource leaks (memory, file descriptors) were solved; sharing and locks between threads were eliminated; C library read functions were replaced by memory-mapping of input files (for up to 10× higher network throughput). The JPX metadata is now sent compressed and several ranges of images can be requested in one JPIP request.

The code is periodically verified with IDEA CLion and Synopsys Coverity static code analyzers and with `valgrind` dynamic analyzer, as well as various sanitize options of several C++ compilers.

[2] https://github.com/Helioviewer-Project/helioviewer.org

[3] https://github.com/Helioviewer-Project/esajpip-SWHV

[4] https://github.com/Helioviewer-Project/GeometryService

[5] https://github.com/Helioviewer-Project/PropagationService

3.2.2 FITS to JPEG2000

The JPEG2000 files can be created with the IDL JPEG2000 implementation. Those files have to be transcoded for the use in the Helioviewer system.

The open-source alternative for the creation of JPEG2000 files is the `fits2img`[6] package, which uses a patched version of the open source OpenJPEG[7] library. The files created by this tool do not need to be transcoded.

While ingesting new datasets during the SWHV project, it became apparent that the metadata in the FITS headers of some datasets is lacking or is defective. A FITS-to-FITS conversion stage is needed for those datasets to adjust the metadata to the needs of the Helioviewer system. At https://github.com/bogdanni/hv-HEP/blob/master/HEP-0010.md there is a summary of those needs.

The following new datasets were added in the course of the project: Kanzelhoehe H-alpha, NSO-GONG far side, NSO-GONG H-alpha, NSO-GONG magnetogram, NSO-SOLIS Azimuth, NSO-SOLIS CoreFlux-Dens, NSO-SOLIS CoreWingInt, NSO-SOLIS FillFactor, NSO-SOLIS Inclination, NSO-SOLIS Intensity 1083Å, NSO-SOLIS Intensity 6302Å, NSO-SOLIS Strength, ROB-USET H-alpha.

In addition, daily radio spectrograms are created from the Callisto network observations. The data files are downloaded from the e-Callisto network website and merged into a composite dataset in order to ensure good 24-hour coverage. The data values are calibrated to correct for instrument sensitivity in frequency and time. During this operation the values are also normalized and transformed to fit into fixed time and frequency bins, covering fixed time and frequency ranges. When multiple values contribute to one bin of the overall image, an average is taken as the final value. An averaging procedure was implemented in order to reduce the noise. It only involves approximately the highest 10% of the signal. This allows reducing the noise sufficiently, while still being able to have enough contrast. The data is written to a temporary FITS file with keywords indicating the frequency and time ranges, frequency and time bin sizes. The composite image is then transformed into a JPEG2000 image file (size 86400×380), one per day.

3.2.3 JPEG2000 files handling

The image data is encoded using the JPEG2000 coding standards. The JPEG2000 data consists of compressed data codestreams organized using *markers* according to a specific syntax, and several file formats, such as JP2 and JPX, which are organized using *boxes* encapsulating the codestreams of compressed image data organized in *packets* and the associated information.

In order to ensure the communication between the server and the client, the Helioviewer system imposes a set of constraints on the codestreams and file formats. This includes requirements for codestream organization such as specific packetization (PLT markers), coding precincts, and order of progression (RPCL), for file format organization, such as the presence of specific boxes aggregating the codestreams and the associated information like metadata, and for file naming conventions.

[6] https://github.com/Helioviewer-Project/fits2img

[7] http://www.openjpeg.org

The JPEG2000 standards have a high degree of sophistication and versatility. In order to encourage the proliferation of Helioviewer image datasets, it should be possible to generate those files with standard conforming software other than the proprietary Kakadu software currently used. It becomes therefore necessary to validate the full structure of Helioviewer image files formally and automatically. A verification system based on Schematron[8] XML schemas was developed. This procedure is able to verify the structure of JPEG2000 file and codestream, including the associated information such as the Helioviewer specific XML metadata, ensuring the end-to-end compatibility with the Helioviewer system.

Before the SWHV project, both the server and the client-side software were derived from the Kakadu Software toolkit[9]. Much of the server-side usage of the Kakadu software can be now replaced with the `fits2img` and `hvJP2K`[10] packages. If the server does not handle files produced by IDL, no server-side use of Kakadu software is necessary.

`hvJP2K` consists in the following tools:

- `hv_jp2_decode`: replacement for `kdu_expand`, sufficient for the web client image tile decoding;
- `hv_jp2_encode`: proto-replacement for `fits2img`, not yet capable of emitting conforming JP2 files;
- `hv_jp2_transcode`: wrapper for `kdu_transcode`, it can output JP2 format and can re-parse the XML metadata to ensure conformity;
- `hv_jp2_verify`: verify the conformity of JP2 file format to ensure end-to-end compatibility;
- `hv_jpx_merge`: standalone replacement for `kdu_merge`, it can create JPX movies out of JP2 files;
- `hv_jpx_mergec`: client for `hv_jpx_merged`, written in C;
- `hv_jpx_merged`: Unix domain sockets threaded server for JPX merging functionality, it avoids the startup overhead of `hv_jpx_merge`;
- `hv_jpx_split`: split JPX movies into standalone JP2 files.

This software is mainly written in Python and is based on the `glymur`[11] and `jpylyzer`[12] open source libraries.

The Helioviewer server needs to interpret JPEG2000 data at several stages:

1. During the generation of the JP2 files – until now typically from software code written in IDL. `fits2img` can replace the Kakadu implementation embedded in IDL. It outputs conforming JP2 files with PLT markers, XML boxes and it has the capability of embedding colormaps.

2. During the ingestion into the Helioviewer server – since the IDL code cannot be configured for the required features, i.e., precincts and PLT markers, the JP2 files produced by IDL have to be transcoded. The vanilla version of Kakadu's `kdu_transcode` program is just a demo application for the Kakadu core system and is not able to produce JP2 files. `hv_jp2_transcode` can wrap the transcoding process and allows to use an unmodified `kdu_transcode`. It is not yet possible to replace the core functionality of `kdu_transcode`. This stage is not necessary for files generated by `fits2img`.

[8] http://en.wikipedia.org/wiki/Schematron

[9] https://kakadusoftware.com

[10] https://github.com/Helioviewer-Project/hvJP2K

[11] https://github.com/quintusdias/glymur

[12] https://github.com/openpreserve/jpylyzer

3. The web client itself has to decode the JP2 files in order to serve the image data to its browser clients, since those do not typically include JPEG2000 support. This can be replaced by `hv_jp2_decode`, at the cost of lower performance, but with little impact for the user experience.
4. Before the image data is streamed using the JPIP server, it has to be aggregated into JPX files. `kdu_merge` can be replaced with `hv_jpx_merge` (up to 10× faster than `kdu_merge`). This task involves only manipulations at the byte level structure of the file formats and not the decoding of the codestreams. From the point of view of the JHelioviewer user, the client – server interaction latency and bandwidth dominate the waiting time for the display of the image data. The server latency has two main components: the database query for the list of files that will make up the JPX and the parsing of those files to extract the information for the assembly of the JPX.

The `hvJP2K` package also contains functionality to enable caching JP2 headers in a database at insertion time and to enable building of JPX files from database headers. The purpose of those two functionalities is to reduce the server latency between the user selecting a dataset and the server making available the prepared JPX file to the `esajpip` server. While merge functionality of `hvJP2K` can be 10× faster than the Kakadu similar functionality, it is limited by I/O operations necessary to parse the JP2 files. Parsing the JP2 ahead of the time and storing the necessary information in a database together with the record of each file can possibly reduce this latency.

This is implemented by two programs part of the `hvJP2K` package (`hv_jpx_merge_to_db` and `hv_jpx_merge_from_db`) which store the extracted JP2 headers into an SQLite3 database and can generate JPX files from those stored headers.

Additionally, the JHelioviewer client has to decode the codestreams for display. The Kakadu software offers superior performance in this area and, for the foreseeable future, will not be replaced on the client side.

3.3 PFSS dataset

A PFSS algorithm[13] was implemented in C for fast computation. GONG magnetograms from http://gong.nso.edu/data/magmap/index.html are used as input. Those FITS files contain a full map of the latest available solar magnetic data at a resolution of 256×180 in a sine-latitude grid.

The PFSS algorithm consists of several steps:

1. For algorithm input, interpolate the magnetogram data onto an appropriate grid;
2. Run the algorithm by solving a Poisson-type equation;
3. Select the field lines and save them to a FITS file.

For the starting points on the photosphere, an equally spaced `theta`–`phi` grid of points that lie above the photosphere is used. The set of starting points is augmented with starting points for which the magnetic field is strong.

[13] http://wso.stanford.edu/words/pfss.pdf

The algorithm to compute the field lines uses an Adams–Bashforth explicit method (third order precision) that requires less evaluations of the vector field than the more commonly used fourth order precision Runge-Kutta methods. This is mainly done because the evaluation of the vector field at a given point is relatively slow.

The resulting FITS files consist of `BINARY TABLE`s with four columns `FIELDLINEx`, `FIELDLINEy`, `FIELDLINEz`, `FIELDLINEs`. The first three are mapped to unsigned shorts and can be converted to Cartesian coordinates using the formula 3 × (value × 2 / 65535 - 1) (on the client side one needs to add 32768). The `FIELDLINEs` value encodes the strength and, by the sign, the radial direction of the field. This encoding was chosen for a compact representation.

The field strength is mapped in the default JHelioviewer display as blue (negative radial) or red (positive radial); the lesser the color saturation, the weaker the field. In order to better see the direction of the field, points of the lines beyond 2.4 solar radii have red or blue colors without blending with white.

4 JHelioviewer interfaces

JHelioviewer communicates with the Helioviewer services using the HTTP network protocol. The JPEG2000 data service is implemented using a subset of the JPIP protocol on top of the HTTP network protocol. In addition, JHelioviewer supports the SAMP[1] protocol and includes a SAMP hub.

4.1 Image services API

The image services API version 2 is documented together with some examples at https://api.helioviewer.org/docs/v2/.

JHelioviewer supports both versions of API and has implemented the validation of server responses against a JSON schema. This functionality is also available as the separate standalone program DataSourcesChecker[2].

The API endpoints used by JHelioviewer are:

- `getDataSources`: retrieve the list of available datasets which is then used by JHelioviewer to populate the "New Layer" UI elements for each server;
- `getJPX`: request a time series of JP2 images as one JPX file;
- `getJP2Image`: to request one JP2 image.

The API server reacts to the `getJPX` call by querying its database of image metadata, constructing a list of filenames to be used, passing the list to the program that will create the JPX file (`kdu_merge` or `hv_jpx_merge`), and making the resulting JPX file available to the `esajpip` server. Finally, the server sends back to the client a JSON response which contains the JPIP URI to use. The client connects to that URI and starts interacting with the JPIP server.

JPX files can be of two types:

- aggregates of metadata with pointers to the JPEG2000 codestream data inside the original JP2 files;
- aggregates of data with embedded JPEG2000 codestreams.

The first form (`linked=true` and `jpip=true`) is used for the regular JHelioviewer interaction over JPIP, while the second form (`linked=false` and `jpip=false`) is used by the "Download layer" functionality to request the assembly of a self-contained JPX file which is then retrieved over HTTP and can be played back on the user computer without a network connection.

The JPEG2000 data services are provided by the `esajpip` server which implements a restricted subset of the JPIP protocol over HTTP (to be described).

[1] http://www.ivoa.net/documents/SAMP/

[2] https://github.com/Helioviewer-Project/DataSourcesChecker

4.2 Timeline services API

The timeline API is a REST service and consists of three parts.

1. **Dataset Query API**: The client requests the description of the available datasets. The description includes group and label names to be used client-side, server-side name and units. It is available at http://swhv.oma.be/datasets/index.php and will list all datasets available and the corresponding groups. The response will be a JSON file with 2 keys:

    - `groups`: the list of groups visible in the client, it has 2 keys:
        - `key`: a unique identifier;
        - `groupLabel`: the name of the group.
    - `objects`: the list of datasets with keys:
        - `baseUrl`: the base URL of where to request the dataset;
        - `group`: the group identifier to which the dataset belongs;
        - `label`: the label of the dataset;
        - `name`: the unique identifier of the dataset.

2. **Data Availability API**: The server returns `coverage` as an array of disjoint time intervals, in increasing order. The coverage intervals are defined as containing data samples no further apart than five times the regular cadence. A similarly defined parameter demarcates the data gaps in the responses to the Data Request API.

3. **Data Request API**: Each individual dataset can be accessed with a URL like:

```
http://swhv.oma.be/datasets/odi_read_data.php?
        start_date=2016-09-01&
        end_date=2016-09-02&
        timeline=GOES_XRSA_ODI&
        data_format=json
```

The first part of the URL is the `baseUrl` from the Dataset Query API. The parameters in the URL are:

- `start_date`: the start date of the wanted timeline in the format YYYY-MM-DD;
- `end_date`: the end date of the wanted timeline in the format YYYY-MM-DD (full day included);
- `timeline`: the name of the timeline as defined by the Dataset Query API;
- `data_format`: only JSON available currently.

The response is a JSON file with the keys:

- `timeline`: the name of the timeline as defined in the Data Request API;
- `multiplier`: the multiplier that needs to be applied to the values;
- `data`: a list of `[timestamp,value]` pairs. The values have to be multiplied by the `multiplier`.

The `multiplier` parameter allows for sending scaled data to the client when necessary. The values of many datasets are rather small numbers when expressed in standard units like W/m^2, thus scaling them allows for more floating-point precision in the response to the client.

The timestamps are with respect to Unix epoch. There is a guarantee that the data is sent ordered by timestamp.

The timeline API is implemented on the server side as several PHP scripts that forward the data requests to the relevant backend timeline storage service and format the JSON response for the JHelioviewer client. The current backend services are based on ODI and STAFF.

4.3 HEK services API

This API is described at http://solar.stanford.edu/hekwiki/ApplicationProgrammingInterface.

A similar API was implemented for the COMESEP alert caching server.

4.4 GeometryService API

The `GeometryService` is a REST network service for solar system geometry computations based on the SPICE[3], SpiceyPy[4], and Spyne[5] toolkits. It can return JSON and MessagePack[6] encoded responses. For example, given the following request:

```
http://swhv.oma.be/position?
            utc=2014-04-12T20:23:35&
            utc_end=2014-04-13T19:44:11&
            deltat=21600&
            observer=SUN&
            target=STEREO%20Ahead&
            ref=HEEQ&
            kind=latitudinal
```

the server returns the following JSON response:

```
{
  "result": [
    { "2014-04-12T20:23:35.000":
      [143356392.01232576,2.712634949777619,0.12486990461569629]},
    { "2014-04-13T02:23:35.000":
      [143359318.57914788,2.7129759257313513,0.12473463991365513]},
    { "2014-04-13T08:23:35.000":
      [143362256.29411626,2.7133174795109087,0.12459673837570125]},
    { "2014-04-13T14:23:35.000":
      [143365205.0945752,2.713659603829239,0.12445620339056596]}
  ]
}
```

This is a list of UTC timestamps and coordinates indicating the geometric position of the STEREO Ahead spacecraft in this example. The first coordinate is the distance to Sun, the second and third coordinates are the Stonyhurst heliographic longitude and latitude of the given object.

The maximum number of points accepted for computation is 1e6 and the Python code distributes the computation to the available number of processors in the computer using Python multi-processing because the SPICE library is not re-entrant.

[3] https://naif.jpl.nasa.gov/naif/

[4] https://github.com/AndrewAnnex/SpiceyPy/

[5] http://spyne.io/

[6] https://msgpack.org

At the moment, the following locations are available: all JPL DE430 ephemeris locations (solar system planets, Pluto, the Moon), comet 67P/Churyumov-Gerasimenko. Also available are the following spacecraft trajectories (existing or planned): SOHO, STEREO, SDO, PROBA-2, PROBA-3, Solar Orbiter, Parker Solar Probe. Several reference frames often used in the heliophysics domain are known.

The following functions are implemented:

- `position` and `state` (in km, km/s, and/or radian) of `target` relative to `observer` in `ref` reference frame, optionally corrected for `abcorr` (`NONE` - geometric, default, `LT`, `LT%2BS`, `CN`, `CN%2BS`, `XLT`, `XLT%2BS`, `XCN`, `XCN%2BS`, see SPICE documentation); representations (`kind`): `rectangular` (default), `latitudinal`, `radec`, `spherical`, `cylindrical`;
- `transform` between `from_ref` reference frame and `to_ref` reference frame; representations (`kind`): `matrix` (default), `angle` (Euler, radian), `quaternion` (SPICE format);
- `utc2scs` and `scs2utc`: transform between UTC and spacecraft OBET (Solar Orbiter supported).

Other arguments:

- `utc`: start of time range;
- `utc_end` (optional): end of time range;
- `deltat` (optional): time step in seconds.

There is a guarantee that the data is sent ordered by UTC timestamps.

This service is used to support the Viewpoint functionality of JHelioviewer.

4.5 PropagationService API

The `PropagationService` is a REST network service which returns JSON encoded responses. It is currently just a mock-up and handles only the case of radial propagation with a fixed speed. It uses the `GeometryService` and is built using the Spyne[7] toolkit.

The following function is implemented:

- `propagate`
  - Arguments:
    * `name` (ignored): quantity;
    * `utc`: start of time range;
    * `utc_end` (optional): end of time range;
    * `deltat` (optional): time step.
  - Returns: rectangular coordinates position of SOHO in HEEQ reference frame in km and fixed propagation speed of 1000km/s.

[7] http://spyne.io/

4.6 SAMP

SAMP capabilities were integrated into JHelioviewer. It is possible to send information about the loaded image layers to SunPy or SolarSoft scripts in order to load the original FITS files via VSO onto the user's computer. It is also possible to receive compatible data from any external SAMP-aware program or web page.

In view of the above SAMP capabilities, direct VSO capabilities are better handled by a community wide effort such as SunPy which incorporates solar physics data web services such as VSO, JSOC and others.

Therefore, the VSO connection is achieved over the SAMP protocol via SunPy or SolarSoft (for which an additional IDL-Java bridge is necessary). Compared with direct VSO capabilities, this solution is better in the sense that the best use of the science data retrieved this way is within an environment made for data analysis such as SunPy or SolarSoft.

The incoming SAMP messages supported by JHelioviewer are:

- `image.load.fits`: specific FITS image;
- `table.load.fits`: only for ESA SOHO Science Archive tool, as JHelioviewer does not support FITS tables yet;
- `jhv.load.image`: any image type supported by JHelioviewer, type determined by filename extension;
- `jhv.load.request`: image request file;
- `jhv.load.timeline`: timeline request file;
- `jhv.load.state`: state file.

Those messages have as parameter an URI to load. The URI argument to `jhv.load.image` can be a list of URIs which will be loaded as a time-ordered movie into one layer. Both local and remote URIs are supported. The URIs supported by `jhv.load.image` can be Helioviewer API calls.

An example of SAMP Web Profile usage is at http://swhv.oma.be/test/samp/.

In addition, JHelioviewer can broadcast information about the loaded image layers. Two clients of this functionality use the information to load the corresponding science data from virtual solar observatories into the SolarSoft and SunPy environments. They are available in examples of the samp4jhv repository.

The broadcasted SAMP message has the following form:

- Message: `jhv.vso.load`
- Arguments:
  - timestamp (string): date of the currently viewed frame coded in ISO8601 format (e.g., 2017-08-28T14:33:28);
  - start (string): start date of the currently viewed sequence coded in ISO8601 format (e.g., 2017-08-28T14:33:28);
  - end (string): end date of the currently viewed sequence coded in ISO8601 format (e.g., 2017-08-28T14:33:28);
  - cadence (SAMP long): number of milliseconds between each frame;
  - cutout.set (SAMP boolean): whether or not only a part of the sun is visible:
    - ⋆ 0: the full Sun is visible;

    * 1: only a cutout of the Sun is visible.
  - cutout.x0 (SAMP float, arcsec, optional): x-position of the currently viewed part of the Sun;
  - cutout.y0 (SAMP float, arcsec, optional): y-position of the currently viewed part of the Sun;
  - cutout.w (SAMP float, arcsec, optional): width of the currently viewed part of the Sun;
  - cutout.h (SAMP float, arcsec, optional): height of the currently viewed part of the Sun;
  - layers (list of map): the different layers currently displayed. The parameters of each layer are stored as a key-value pair with the following keys:
    * observatory (string, required);
    * instrument (string, required);
    * detector (string, optional);
    * measurement (string, optional);
    * timestamp (string, ISO8601 date, required).

    The keys which are set depend on the selected instrument.

- Return Values: None.
- Description: Broadcasts information about all the currently visible layers in JHelioviewer including the current timestamp. Receiving applications can use this information to load the raw data from VSO, for example.

Example:

```
{
  samp.mtype=jhv.vso.load,
  samp.params={
    timestamp=2017-09-24T19:52:28, start=2017-09-24T00:00:00, end=2017-09-26T00:00:00,
    cutout.set=1, cutout.h=2460.524544, cutout.w=2460.524544, cutout.x0=3.3579912600000625,
        cutout.y0=1.1773994400000447,
    cadence=1800000,
    layers=[
      {observatory=SDO, instrument=AIA, detector=, measurement=304, timestamp=2017-09-24T19
          :53:05},
      {observatory=SDO, instrument=AIA, detector=, measurement=171, timestamp=2017-09-24T19
          :52:45},
      {observatory=SDO, instrument=AIA, detector=, measurement=193, timestamp=2017-09-24T19
          :52:28}
    ]
  }
}
```

4.7 File formats

Many of the file formats supported by JHelioviewer are based on the JSON format. All files can be either local on the user's computer or can be loaded over HTTP. JPX data can additionally be loaded over the JPIP protocol. Files can be loaded at start-up via the command line interface.

4.7.1 State file

This is a JSON document which can be saved and loaded from the UI. It preserves with high fidelity the state of the program. Most of the fields have direct correspondence to the user interface, thus they are self-documenting. Natural language specification for time is supported. Most of the fields are optional with sensible defaults.

An example file is too long to be included here and is maintained at https://github.com/Helioviewer-Project/JHelioviewer-SWHV/blob/master/extra/example_state.jhv.

4.7.2 Image request file

JSON document specifying an array of image requests to the default server in a simple manner as in the example below. Natural language specification for time is supported and, besides the `dataset` field, all fields are optional with sensible defaults.

Example:

```
{
  "org.helioviewer.jhv.request.image": [
    {
      "observatory":"SDO",
      "startTime":"yesterday",
      "endTime":"today",
      "cadence":1800,
      "dataset":"AIA 304"
    }
  ]
}
```

4.7.3 Timeline request file

Example:

```
{
  "org.helioviewer.jhv.request.timeline": [
    {
      "timeline": "GOES_XRSA_ODI"
    }
  ]
}
```

4.7.4 Image formats

WCS metadata is used to place image data at the correct viewpoint (time and position). Without metadata, image data is placed at a default viewpoint. JHelioviewer can extract and interpret metadata from JP2, JPX, and FITS formats.

JHelioviewer can extract Helioviewer XML metadata inserted by `fits2img` into PNG and JPEG image files via the PNG text chunk and JPEG comment marker mechanisms. It also supports at a basic level, without metadata, the Java ImageIO formats (JPEG, PNG, BMP, GIF).

JHelioviewer can support CDF format using the https://github.com/mbtaylor/jcdf library and VOTable format using the https://github.com/aschaaff/savot library.

The FITS support include compressed and remote files, physical units, the BLANK keyword, pixel scaling including the port of ZMax autoscaling algorithm of SAOImage DS9 (better results for EUV observations), and gamma-correction and logarithmic transfer functions. JHelioviewer can display the value of the pixel under the mouse pointer in physical values.

This support is limited to known use cases exhibiting data calibration and known standard metadata.

4.7.5 Command-line interface

The following command-line options are available:

```
-load    file location
       Load or request a supported file at program start. The option can be used multiple
           times.
-request request file location
       Load a request file and issue a request at program start. The option can be used
           multiple times.
-state   state file
       Load state file.
```

5 JHelioviewer design

In contrast to the 32k lines of code to implement all its many features, the core JHelioviewer design is very simple and can probably be expressed in a couple of thousands of lines of code.

The following figure presents an outline of the JHelioviewer architecture.

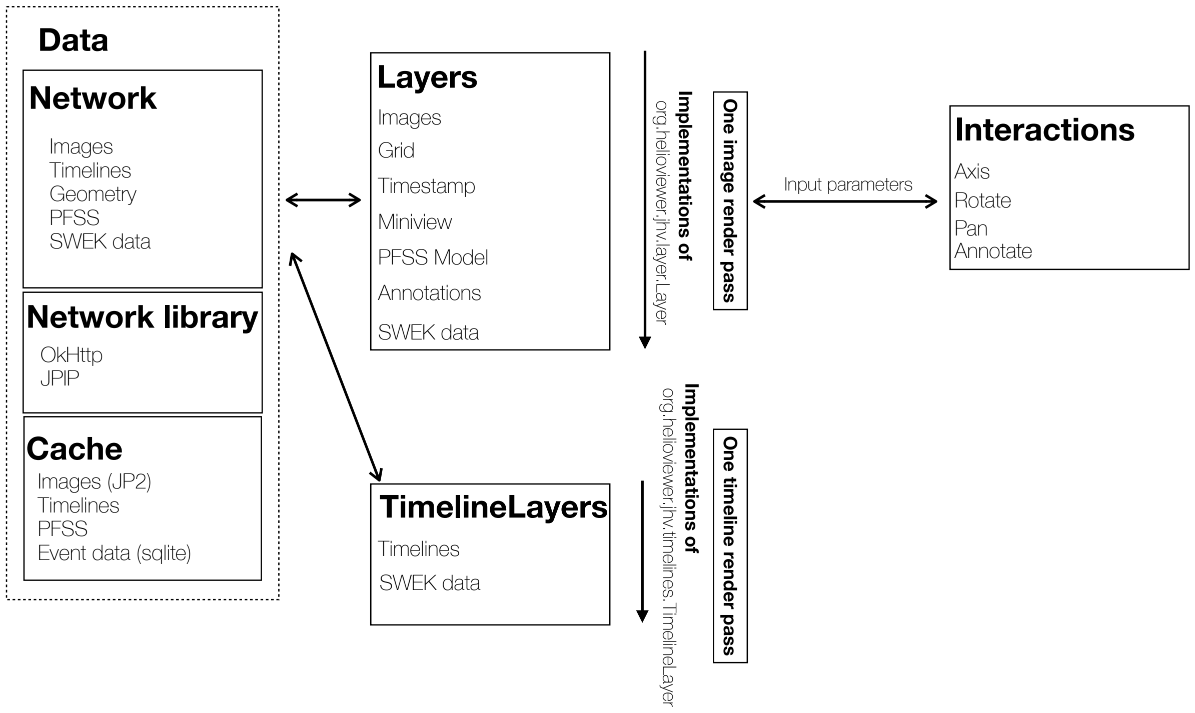

Figure 5.1: JHelioviewer architecture outline

The program is structured in a manner that is amenable to performance. The principle of separation of concerns is applied throughout. Objects are asked to update themselves, they proceed to do so independently, and they report back when done. To remain responsive while performing long lasting network and computation operations, the program uses threads, caches, and high performance algorithms and data structures. There are essentially no locks and few data structures are accessed from concurrent threads.

JHelioviewer operates in a fully asynchronous manner and can achieve high frame rates with low CPU utilization while remaining responsive to interactive user input.

The program is driven via two timers:

- `Movie` beats at a configurable frequency (default 20 Hz) and commands the setting of the program time (i.e., frame advance);
- `UITimer` beats at constant 10 Hz and commands the refresh of the Swing UI components that need to change together with the movie frame; additionally, it commands the refresh of the timeline canvas.

Various parts of the program can request to refresh the image canvas and a balance has to be found between avoiding excessive redraws and avoiding CPU wake-ups when idle. `Interaction` contains a supplemental timer which is started on-demand and is used to limit the rate of user-induced zoom requests leading to decoding and redraw requests.

5.1 Concepts

5.1.1 Time

The image timestamps are handled as opaque `JHVTime` objects which express them as milliseconds since the Unix epoch. `JHVTime` can also be expressed as string for display in the user interface, being converted on demand and cached by a high-performance cache. The pervasive use of `JHVTime` enables the introduction of supplementary time reference frames besides the observed time, such as Sun center view time or Earth view time, in order to facilitate the comparison of observations from different solar system vantage points.

5.1.2 Coordinates

Reference frames orientations are computed using SPICE via the JNI mechanism for foreign function calling of native compiled code interfaces. A minimal set of SPICE kernels expressing positions of solar system objects and definitions of reference frames often used for the analysis of solar physics data is included with the JHelioviewer resources JAR. Supplementary trajectories and reference frames transformations for artificial satellites are obtained over the network from the `GeometryService`. Therefore, those can be updated independently from the program.

Distances are expressed in units of solar radii (photometric, Allen). This is for numerical stability and to ease the expression of some computations. Additionally, this unit helps with the limited precision of the OpenGL depth. On ingestion, the program can optionally normalize the apparent solar radius observed in various EUV wavelengths to the photometric reference. This is done to match the synthetic elements drawn independently, such as the grid, to the data. The program is able to display elements at distances up to about $10755R_\odot$ from the Sun, which is beyond the aphelion of Pluto.

Orientation is expressed in latitudinal form (latitude, longitude in radian) with respect to a Carrington reference frame for ease of expression of rotations as quaternions. Computations of rotations are performed using quaternions for performance and numerical stability reasons. The interaction with OpenGL is done using matrices since, besides rotation, it involves projection and translation.

Together with a timestamp, the distance to Sun and the orientation constitute the fundamental concept of `Viewpoint`. One important viewpoint is Earth's.

Viewpoints can be computed at various timestamps using the `UpdateViewpoint`. Several forms are provided:

- `Observer` takes the closest in time from the metadata of the master layer, see the Metadata section.
- `Earth` computes the viewpoint with the algorithm mentioned above.
- `EarthFixedDistance` is as above but with the distance fixed at 1au (it is used for the latitudinal and polar projections).
- `Equatorial` is a viewpoint looking from above the solar north pole at a distance (~229au) such that, for a field-of-view angle of 1°, objects up to 2au far from the Sun are visible, the longitude is derived from the closest in time metadata of the master layer such that the Earth appears on the right-hand side.
- `Other` (in UI, `Expert` in code) are viewpoints which are computed from the responses to requests to the `GeometryService`. A maximum of 10000 points are requested to the server and interpolation is used as needed for the intermediate timestamps.

5.1.3 Camera

The computed `Viewpoint` is used in setting up of the `Camera`, which intermediates to the rest of the program and to OpenGL.

One important task of `Camera` is to set up the projection matrix. The projection is always a variant of an orthographic projection with the Sun at depth 0. In the orthographic display mode, there are two types of projection matrices, one with deep clipping planes (range [-10755$R_\odot$,+10755$R_\odot$]), appropriate for far viewing distances such of the `Equatorial` viewpoint, and one with more shallow clipping planes (range [-32$R_\odot$,+32$R_\odot$], a bit more than LASCO C3 FOV), appropriate for the normal solar observations. This duality is necessary for the preservation of precision in the OpenGL depth buffer.

Another important task of `Camera` is to set up the model-view matrix. This is based on the orientation of the `Viewpoint` and on the rotation and the translation due to the user interaction with the image canvas. When image data draw commands are issued, a difference rotation with respect to the image metadata is computed, configured and used in the shader programs. For other drawn elements, the camera orientation may be saved, the camera may be rotated as desired, the draw command issued, and then the camera orientation may be restored.

Functionality to translate from the two-dimensional coordinates of the image canvas to the three-dimensional internal coordinates is available in `CameraHelper`.

5.1.4 Metadata

Metadata about the observations is extracted from the incoming image data format. It is currently possible to extract it either from the JPEG2000 formatted streams or from the FITS formatted streams. In the JPEG2000 streams it was derived from the original FITS header and inserted as XML at the moment of creation of the JP2 and JPX files while, for FITS streams, it is part of the format and it is transformed by the program to XML in order to use the same parsing code.

A discussion about the necessary metadata components is at https://github.com/bogdanni/hv-HEP/blob/master/HEP-0010.md.

If the necessary metadata can be derived, it is made available to the rest of the program in `HelioviewerMetaData` structures. For image formats without metadata, or when the metadata is not present, or its parsing fails, a default metadata structure is built corresponding to the Earth viewpoint at 2000-01-01T00:00:00.

5.1.5 View

A `View` is a representation of the incoming image data stream which knows how to read the stream and to decode it into pixel data which can be drawn. Several `View`s can be bundled together into a `ManyView` with the same interface, in order to allow the playback of disparate frames as a time-ordered movie. `ManyView` uses a high-performance data structure binding the `JHVTime`s of all subordinate frames in order to enable efficient query of frames timestamps.

There are several specializations:

- `URIView` decodes image streams using Java `ImageIO` API and FITS streams using the `nom-tam-fits`[1] library.
- `J2KView` deals with JPEG2000 streams and it is the most complex, as it has to implement both the image decoding via the Kakadu library and to implement the JPIP streaming protocol.

`J2KViewCallisto` is a specialization of `J2KView` which uses the `NetClient` interface to read JP2 files over HTTP, then cache and decode them as local files.

`URIView` and `J2KView` issue commands to decode image data over a queue of size one to the `DecodeExecutor` which handles just one thread at a time. If a command is already queued, it is removed from the queue to make place for the most recent one.

Both `URIView` and `J2KView` cache the decoded pixel data as `ImageBuffer`s in Java memory. If the requested image data corresponds to an item in the cache, the item is immediately returned instead of entering the decode processing. The cache uses soft references, thus the decoded pixel data fills the JVM heap which has a fixed size established at program startup. When the maximum heap size is reached, the garbage collector eagerly collects the buffers not referenced elsewhere in the program. The effect is an automatic trade-off between memory and CPU usage for already decoded pixel data and the cache adaptation to the JVM heap size. The benefits scale with the available JVM heap and with the number of loaded image layers.

In the case of JPEG2000 data, this approach also leads to a potentially dramatic drop for the total memory used by the application since the JHelioviewer-side caching uses one byte per pixel (for color-mapped layers) compared to the four bytes per pixel in the cache of `Kdu_region_compositor`. Another benefit is a performance increase due to reduced need for copies between Kakadu native buffers and JVM memory.

The decoded pixel data together with the associated information such as the metadata is constructed into `ImageData` structures which are handed over to the `ImageDataHandler`, i.e., `ImageLayer`.

Some experiments with texture data upload into OpenGL directly from the decoding thread were conducted. Such an approach would shift the decoded image data cache from the JVM heap to OpenGL, i.e.,

[1] https://github.com/nom-tam-fits/nom-tam-fits

process memory or texture buffers inside GPU memory, as decided by OpenGL. The disadvantages are: duplicated code paths for radio data which needs the image data into Java memory, difficult OpenGL buffers handling, and more sophisticated OpenGL context handling. The latter can be approached by migration of the OpenGL context between threads or, alternatively, multiple contexts. The first approach leads to thread contention for the context, which blocks the rendering pipeline and thus may introduce stuttering at high frame rates, while the second approach leads to OpenGL context switch overhead. The experiments concluded that this may be useful only for many loaded layers.

The two major components of `J2KView` are:

- `J2KReader` implements a minimal HTTP client and a minimal JPIP-over-HTTP streaming client. It fills the memory cache of `JPIPCache` (an extension of Kakadu `KduCache`) from which the image decoding takes place, see the Persistent JPEG2000 data section. Once the entirety of the data for a resolution level for a frame is available, it is extracted from the memory cache and it is sent via `JPIPCacheManager` into the disk persistence layer provided by the `Ehcache` library. The `sourceId` of the dataset and the timestamp of the frame is combined to form universal identifiers. `J2KReader` is also used to fill the cache indicator of the UI time slider via the `CacheStatus` interface. `J2KReader` is implemented as a thread that receives from `J2KView` commands to read. It tries to read requested data first via the `JPIPCacheManager` before constructing and issuing a request to the JPIP server if not available in the persistent disk cache. Once all data is read, the HTTP connection is closed and the `J2KReader` stops listening for commands.
- `J2KDecoder` is in charge for forwarding commands to `Kdu_region_compositor` to decode image data out of `JPIPCache`. A `Kdu_thread_env` object in thread local storage is used to distribute the decoding work of the native code over up to 4 CPUs. The resulting bytes of `Kdu_compositor_buf` in native memory are copied one-by-one using LWJGL[2] `MemoryUtil.memGetByte`, which was found to have the highest performance and to reduce the size of necessary memory buffers.

5.1.6 Layer

A `Layer` is an interface to an object that knows how to fetch its data and to draw itself on the image canvas. The program orders them in a `Layers` list and represents them in the user interface via a `LayersPanel` list selector where the user can interact with them (add, remove, select, make invisible). The layers in the list are drawn in order from the top to the bottom of the list selector. Each layer has a panel of options visible under the list selector when the layer is selected. Those options are used to configure the draw commands.

A special group of layers is made of `ImageLayer`. Those can be re-ordered in the list by the user via drag-and-drop. Additional layers include `ViewpointLayer`, `GridLayer`, `FOVLayer`, `TimestampLayer`, `MiniviewLayer`, `SWEKLayer`, and `PfssLayer`. The names correspond to user visible functionalities.

The most sophisticated type of layer is the `ImageLayer`. This is because it implements the core functionality for image display and because it allows for the replacement of the underlying `View`. It manages the `DecodeExecutor` thread pool on which the image decode activity of `View` is performed.

[2] https://www.lwjgl.org

The same concept is used for timelines, where several of `TimelineLayer` are organized in a `TimelineLayers` list and are represented in the user interface via the `TimelinePanel` list selector. Specializations are `Band` for plots, `RadioData` for Callisto spectrograms, and `EventTimelineLayer` for events.

5.2 Time handling

Time selection for the image layers was brought to the forefront and it became possible to use the interaction with time range of the timelines panel to temporally navigate jointly with the image layers. This is achieved by reloading as necessary the image layers for the time range of the timelines panel. The capability to cache the JPEG2000 codestream data improved the user experience.

5.3 Drawing

JHelioviewer uses alpha-premultiplied color blending. The default compositing of the layers is at the middle between the ADD and OVER operators and a per-layer setting named "Blend" controls the variable level of additivity and opacity.

The image canvas is implemented using a JOGL[3] `GLCanvas` which integrates into the rest of the Swing interface. To handle the situation in which the OpenGL context has to be recreated due to events in the underlying windowing system, the canvas uses a slave context derived from a master context of a dummy drawable. `GLListener` implements the event based mechanism for performing OpenGL rendering (`init()`/`dispose()`/`reshape()`/`display()`). All rendering ensues from `display()` in the AWT Event Dispatch Thread and uses the primary rendering context associated with the `GLCanvas` for its lifetime. Similarly for initialization and disposal.

The drawing on the image canvas is done entirely using GLSL programs. The following interfaces are available:

- `GLSLSolar` handles `solarOrtho`, `solarLati`, `solarPolar`, and `solarLogPolar` shaders to draw image data, and it is used exclusively by `ImageLayer`. Those shaders implement the projected drawing and optionally can distort the drawing according to solar differential rotation (Snodgrass, magnetic features).
- `GLSLLine` handles the `line` shaders to use instanced rendering for drawing triangles out of line segments.
- `GLSLShape` handles the `shape` and `point` shaders to draw polygons and points.
- `GLSLTexture` handles `texture` shaders to superimpose texture data such as event icons, and it is used by `JhvTextRenderer` to draw text rendered into cached textures.

Besides the drawing done by layers, annotations can be drawn by `InteractionAnnotate`. On solar surface, rectangle, circle, and cross annotations can be drawn. Arbitrary FOV annotations can be drawn and their centers are indicated. There is functionality to automatically zoom to the size of the current FOV annotation.

[3] https://jogamp.org

Matrices compatible with the OpenGL representation are maintained by the `Transform` class which implements projection and model-view matrix stacks and a simple cache of the result of the multiplication of the top of the matrix stacks. The stacks can be pushed and popped similarly to the traditional OpenGL fixed-function matrix stacks and are queried at configuration time by the draw commands. This is implemented using the JOML library.

5.4 User input

The user input (mouse movements and keyboard strokes) to the image canvas is received by the `GLCanvas` from the JOGL input system. All computations in response to user interaction are performed on the Swing thread. The event dispatch to the interested subscribers is mediated by the `InputController` which also scales the input mouse coordinates to OpenGL pixel scale (possibly HiDpi). The available interactions are

- `InteractionAnnotate`: draw annotations, possibly interactively, on the image canvas;
- `InteractionAxis`: rotate around the current `Viewpoint` axis;
- `InteractionPan`: translate camera origin;
- `InteractionRotate`: free rotation around camera origin.

5.5 Network I/O

Besides the implementation specific for the JPIP network client, all the rest of network I/O is done via `NetClient`, which provides an interface implemented on top of Okio[4] and OkHTTP3[5], transparent to the actual location – remote or local – of the requested resource. All the remote APIs are REST and the implemented functionality is that of the HTTP GET request. The caching functionality of OkHTTP3 can be used or can be bypassed. An additional local cache where direct access to the cached files is possible is provided by `NetFileCache`. Those caches are specific for each program instance and are removed at program exit.

Re-implementing the JPIP protocol on top of the high-level OkHTTP3 library appears more difficult. It is not clear if a re-implementation is beneficial in the context of how the `esajpip` server handles the connections with the clients. The current low-level access into the network software stack is useful for achieving high data throughput between the JPIP client and server.

Proxy support is implemented using the Java network proxy facilities.

5.6 Caches

The decoded pixel data caches were mentioned in the View section and the network responses caches in the Network I/O section.

[4] https://square.github.io/okio/

[5] https://square.github.io/okhttp/

JHelioviewer includes several other memory cache mechanisms:

- radio data objects (7 days)
- decoded PFSS data (soft references)
- string representations of `JHVTime` (100k strings)
- Earth positions in Carrington and HCI frames (10k each) indexed by time

The special case of persistent JPEG2000 data is described in the following section.

5.6.1 Persistent JPEG2000 data

This topic deals with the persistent disk caching of JPEG2000 data and is intimately linked with the playback and how the JPIP protocol is handled. Having the capability of reusing already downloaded data is highly desirable. This reuse can be between program runs, or between streams, for example when extending or shifting the temporal navigation of datasets.

JHelioviewer abandoned the concept of downloading regions of interest from the JPIP server, but still tries to minimize the resolution levels requested, such that, if the highest resolutions are never requested, they are never downloaded. Note, however, that recording movies forces the decoding of the full resolution of frames and therefore the download of the entire codestream.

JHelioviewer implements JPEG2000 data disk persistence by extracting the codestream data from the Kakadu `KduCache` and then by storing it using Ehcache3[6]. The process is performed as needed after the data for a resolution level of a frame is completely received. Higher resolution levels overwrite lower ones. The caching is cross-server under the assumption of immutability and idempotence (Callisto spectrogram data is treated differently, it is requested over HTTP and it is cached at the level of JP2 files). The `sourceId` and timestamp of observation are combined to generate unique identifiers. The memory overhead is minimal because only the disk cache tier of Ehcache is used.

The memory codestream cache is handled by Kakadu's `KduCache`. Given the cache tiers (memory, off-heap, disk, and even clustered) capabilities of the Ehcache 3 library used, it is probably possible to restrict the size of `KduCache` objects. However, JHelioviewer was demonstrated to be capable of playing back streams of 10000 frames. This is the limit accepted by the ROB server, GSFC and IAS servers are limited to 1000 frames per stream.

5.7 Movie export

The movie export is enabled by off-screen rendering in an OpenGL framebuffer object. The advantage of this approach is that the output frames can have different aspect ratios than the image canvas and they do not have to be scaled for the desired size. The scene rendering is performed twice, first on the framebuffer object and second on the regular image canvas. The rendered pixel data is extracted from the texture attached to the framebuffer object and buffered into a memory-mapped byte buffer, in order to avoid running out of Java heap memory during long recordings at high framerates. This also enables byte level address manipulation. The buffered frames are flipped (OpenGL uses a lower left coordinate

[6] http://www.ehcache.org

system) and optionally pasted together with the timelines canvas capture and prepared in a temporary file appropriate for ffmpeg[7] input. Once the recording stops, the program spawns an encoding ffmpeg instance. Once the ffmpeg process exits the transient files are removed.

5.8 Timelines

An important insight for high performance plotting is that plots can be requested for short and long time ranges, and, for the latter case, an excess of data points make for an illegible plot. Therefore, a multi-resolution approach for the time dimension is beneficial.

For the Callisto spectrograms which are 2D, this approach is supported by using the JPEG2000 format for the data, with daily images having on the x-axis the second of the day, and on the y-axis the frequency bin.

The 1D timelines are cached in data structures which are pyramids of time resolutions. The highest resolution has one-minute bins and the lower resolutions are obtained by decimation. Depending on the nature of the dataset, several types of data are supported, namely linear, strictly positive linear, and logarithmic. For the latter two, the time decimation is performed by taking the maximum in the time bin to be decimated, as the user is likely to be interested in data peaks, e.g., flares.

5.8.1 Propagation of in-situ measurements

The considered use-case is measurements of phenomena propagating radially slower than speed of light.

This is how one user imagines this feature would work:

> The user sees in the SWAP movie a flare and dimming on the Sun and wonders if this has arrived at the Earth/L1. They open a GOES X-ray timeline to see the flare timing and then open an ACE solar wind timeline to see the arrival. They will now put in the text box reasonable differences from 20h to 5d and see if there is a match of the flare timeline and of the solar wind disturbance. A slightly more natural way to do this would be to put in propagation speeds. From the SWAP movie one can get at least a rough feeling if this is a slow eruption (300 km/s) or a fast one (2000 km/s). The user fills in a guess for a speed, the system calculates the delta-t in seconds and we see the two timelines.

The timeline options panel has a text box where one can set a propagation speed (0 meaning disabled). The location is the calculated Sun-Earth L1 point. When that speed is different from zero, another time-scale (in the timeline color) appears under the timelines panel time-scale (in black).

Therefore, the displayed time-scale has the following meaning:

- the colored time-scale is time of observation;
- when speed is 0, the time of the timelines panel (black time) is disconnected from image layers time → time is UTC at an undefined location (most likely Earth);

[7] https://ffmpeg.org

- when speed is not 0, timelines panel time (black time) is the time determined by the image layers viewpoint, there is an additional speed-of-light propagation from Sun to the viewpoint → time is UTC at the viewpoint (according to viewpoint settings, may be different from Earth).

Several DSCOVR in-situ timeline datasets identified as highest priority for the space weather forecasters were integrated. The added datasets are the interplanetary magnetic field `Bz`, `Bt`, and `Phi` and the solar wind density, radial speed, and temperature.

5.9 Events

Supported event sources are HEK and COMESEP. The interface to the COMESEP caching server is similar to the interface to the HEK and the format was designed to be similar, namely JSON structures with parameters. Since the user can view a large variety of events over long time ranges, significant effort was put into reducing the memory usage as Java objects have a high memory overhead. This was done by pruning some of the parameters and extensive use of interned strings.

The event data is cached into an SQLite database, with the JSON structures inserted as gzip-compressed blobs. The last two weeks' worth of data can be updated to allow for server-side corrections. Some of the parameters are also available for query, allowing to implement filtering of some event types on the value of those parameters.

5.10 Plugins

Historically, plugins were implemented as standalone libraries to be loaded by the program to augment functionality. They necessitated a relatively limited support from the main program. As the functionality of the program becomes more complex and the expectations of integration between the parts higher, it is unclear if this approach is sustainable and if keeping programming interfaces too stable impedes progress. The current plugins are so integrated with the main program that they represent merely GUI elements which can be optionally disabled for a more streamlined appearance.

As a consequence of implementing the SAMP functionality, JHelioviewer has simple programming interfaces for adding data layers and for loading state.

5.11 Compilation and installation

The `lib` directory in the source tree contains the libraries necessary for the compilation and execution of the program in Java JAR format. Seldom changing resource files such as fonts, images, colormaps, SPICE kernels, licenses, and settings are collected into a separate JAR file. Native libraries such as ffmpeg, SPICE, and Kakadu are separated in JAR files per operating system.

The support for 32bit operating systems was removed in order to avoid the incidence of out-of-memory crashes and virtual address exhaustion, especially during movie creation. The software is supported under Windows, macOS and Linux for 64bit OpenJDK or Oracle Java, versions 11 or later.

The program can be built using `ant` or using the `gradle` integration with `ant`. The `release` directory contains another `ant` build script for the preparation of the distribution JARs: the JHelioviewer Java code and collections of native libraries extracted for each operating system, to be bundled separately in order to reduce the size of the distributed packages.

The distribution packages are built using install4j[8] driven by a configuration file part of the source tree. It can generate native launchers for each supported operating system and installation packages in the following formats:

- DMG for macOS containing JHelioviewer.app (optionally signed and notarized);
- EXE installation wizard for Windows;
- DEB and RPM for Linux.

For Windows and macOS, install4j bundles a tailored JRE distribution out of OpenJDK 11, therefore the users do not need a dedicated Java installation.

The distribution packages are copied manually to http://swhv.oma.be/download/ alongside a `VERSION` file which, at program startup, is compared with the builtin version in order to determine the latest available version and possibly alert the user about a newer release.

The version number is in the `MAJOR.MINOR.PATCH.REVISION` format.

The software is under continuous refactoring and re-architecting as new features become available. Simplification and reduction of the number of lines of code are top priorities. Besides continuous testing, the software is regularly submitted to static code analysis using IDEA IntelliJ, Google ErrorProne, SpotBugs, PMD, and Synopsys Coverity.

5.12 Telemetry

JHelioviewer communicates back only crash reports to a Sentry server deployed at ROB. This server is connected to private email, Slack and GitLab services. Since the 2.12 release, several crashes unreported by users were intercepted. Some which were not uncovered by testing were fixed, others need more information.

The best manner to report issues is via https://github.com/Helioviewer-Project/JHelioviewer-SWHV/issues.

The contributing guidelines are available at: https://github.com/Helioviewer-Project/JHelioviewer-SWHV/blob/master/CONTRIBUTING.md.

5.13 JPIP alternatives

5.13.1 Context

JHelioviewer is a Java application that requires download and installation on a local machine. A (possibly lighter) version on the Web would reach the masses that aren't willing or cannot make the step

[8] https://www.ej-technologies.com/products/install4j/overview.html

to install a specific software package. Furthermore, web-based access to JHelioviewer could provide its functionality also to mobile devices. However, this requires an approach based on a JPIP alternative, as JPIP decoding is currently difficult to be implemented in a web-browser. In this context, it was important to investigate how to provide web-access to JHelioviewer functionality using alternatives to JPIP. The work focused on video streaming, as this is the most serious challenge to solve.

For this WP, a web-based video streaming prototype has been investigated and implemented. It serves as a proof-of-concept to show in which direction web-based streaming of SDO data could go. Our implementation could also pave the way to include “movies” from another spacecraft, including Solar Orbiter. The work focused on the identification and verification of the required technical core components.

The prototype made available to this project is an enhanced version of the code delivered in 2017. It enhanced significantly the frame rate as part of a student project. We provide this version as it represents what we believe to be the best web-based video streaming demonstrator of SDO data available at the moment.

5.13.2 Prototype and code location

A proof-of-concept prototype exists locally at FHNW and shall be made publicly available soon. The full code base is available at https://github.com/i4Ds/Scalable-Video-Streaming/tree/js-decoding and there is also an earlier approach which contains both the 2017 and the student-enhanced code at https://github.com/heliostreamer.

5.13.3 High-level technological concepts

Video support: The most supported video format supported by current web browsers is MP4 H.264 (https://caniuse.com/#feat=mpeg4), including “fragmented” versions for streaming. All tested platforms support HTML5 video playback of grayscale 8bit interlaced MP4 videos.

Storage: For the prototype, a video tree has been built from AIA images, covering one day, with 4 different pixel resolutions (from 4k full resolution down to 512) and 6 different time resolutions, i.e., playback speeds (1×, 2×, 4×, 8×, 16×, 32×). Helioviewer.org provided 2379 JPEG2000 images, from which 6035 videos were built. With the current video format, we expect 50TB of videos, compared to the 57TB of JPEG2000 images they cover.

Bandwidth: In full resolution streaming mode, i.e., streaming the full 4k images, only 7MB/sec are required. Yet the tool is built to download with region of interests, with a realistic worst case of 2.5MB/sec bandwidth required. For mobile devices, bandwidth usage can be throttled down to 0.15MB/sec.

WebGL: The prototype projects 64 videos (8 512px videos in width and height) onto a 3D sphere. While the GPU has no issues, this worst-case performance test shows significant CPU consumption, with the naive implementation slowing down the playback to 10 frames per second. We are confident though that this can be addressed with improved GPU communication. Practically all devices support WebGL and 4k textures (https://webglstats.com/webgl/parameter/MAX_TEXTURE_SIZE).

Decoding: The decoding speed of MP4 videos proved to be no performance bottleneck at all. Even more, this work can be offloaded to a worker thread, and would therefore be non-blocking. All browsers support web workers (https://caniuse.com/#feat=webworkers).

5.13.4 Limitations of the 2017 prototype

The design of both the 2017 and the student-enhanced prototype requires a high-end laptop or a desktop to play the videos fluently. Low-end machines or portable devices will require further considerations.

The prototype displays and synchronizes at most 9 video tiles at the same time, because the synchronization is expensive for HTML5-Video elements. However, some viewing angles would require more than 9 tiles to be visible, leading to paused tiles on the sphere. This happens, for instance, when viewing the sphere from the side.

Not all web-browsers behave in the same way. The currently supported browser is Chrome, and further work would be needed to support all the other browsers.

Currently, only short periods of video are available for each playback speed.

5.13.5 Changes in current prototype (locally at FHNW)

As it was shown in the 2017 prototype, the decoding itself is not the bottleneck. Therefore we explored a new approach which uses a javascript decoder to stream and decode multiple videos simultaneously and render them to a HTML5 canvas. This way the synchronization of the different videos comes at almost zero cost as there is full access to the single video frames. It was shown that this way even a tablet can display the prototype, at least for a short video sequence.

5.13.6 Limitations of the current prototype

Several features such as 3D mapping onto a sphere and choosing different speeds were not yet implemented in this version, but we are confident that those do not add a substantial amount of CPU power. While on a desktop the bottlenck for this approach is mainly the bandwith, decoding is only an issue when using a mobile device such as a tablet. This could be further optimized by parallelization (worker threads).

5.13.7 Conclusion

The prototype showed that from this first evaluation no criteria prevents further investigation into video streaming of AIA images. The current prototype also proved that streaming AIA images onto a mobile device is possible.

Furthermore, with the growing support for the Media Source Extensions API (https://caniuse.com/#feat=mediasource), close control can be exercised over video playback, chunking and streaming while having the benefits of HW-decoding and hence significantly improved decoding speed on mobile devices. For caching, `IndexedDB` seems suitable (https://caniuse.com/#search=indexeddb).

6 CCN3 tasks

Several tasks have seen significant progress at this stage of the CCN3 activity. This is visible in the Traceability matrix.

6.1 Task 3: Perform updates on the JHV client-side

6.1.1 Implement selected user requirements for Solar Orbiter mission, including support for low-latency, quick-look and science data (WP30300)

Several user requirements have been implemented. Remaining requirements do not require changes in the design, but the new capabilities will enable internal code restructuring for increased clarity.

6.1.2 Improve time handling inside the program by making it independent of the time stamps of movie frames (WP30300)

The basis of this change was already implemented together with the changes to enable the movie playback speed expressed in solar time/second feature. Setting the time range independently from the movie frames was implemented by introducing of an invisible image layer which is active when no image layer is loaded.

6.1.3 Improve performance of movie export (WP30300)

This has been implemented by the change away from the Java-based movie export library to ffmpeg, see the Movie export section. This also enabled the configurability of the output format, as well as the lossless export in form of PNG file sequences.

6.1.4 Increase UI consistency, clarity and functionality (WP30300)

Several improvements were introduced such as drag-and-drop capabilities for image files, non-modal annotations, consolidation of program preferences, the simplification of datetime selection as well as the capability for SPICE time formats and NLP time input. Subsequent improvements will be made possible by capability to set the program time and timestep independently from the loaded layers.

6.1.5 Improve the installation procedures for a modular, customized Java distribution (WP30300)

This has been implemented by updating the code to use Java 11 interfaces and by using the install4j procedure for customized Java distribution, see the Compilation and installation section.

6.1.6 Make JHV more versatile by supporting additional data formats (WP30300)

This has been implemented by allowing to insert and extract Helioviewer metadata in JPEG and PNG formats, see the Image Formats section. Additionally, individual frames can now be played-back as a time-ordered movie.

To facilitate the visualisation of model outputs, a preliminary specification of a file format inspired by GeoJSON[1] was introduced. This will allow JHV to draw points and line segments throughout the heliosphere.

An example below:

```
{
    "type" : "SunJSON",
    "time" : "2021-11-11T11:11:11",
    "geometry" : [
        {
            "type" : "ellipse",
            "coordinates" : [[2, 60, 0], [1, 60, 90], [3, 60, 0]],
            "colors" : [[255, 0, 0, 255]],
            "thickness" : 0.004
        },
        {
            "type" : "line",
            "coordinates" : [[1.1, 135, 45], [1.2, 135, 45], [1.3, 135, 45], [1.4, 135, 45]],
            "colors" : [[255, 255, 255, 255], [255, 0, 0, 255], [0, 255, 0, 255], [0, 0, 255,
                255]],
            "thickness" : 0.016
        },
        {
            "type" : "point",
            "coordinates" : [[1.1, 45, 45], [1.2, 45, 45], [1.3, 45, 45], [1.4, 45, 45]],
            "colors" : [[255, 0, 0, 255], [0, 255, 0, 255], [0, 0, 255, 255]],
            "thickness" : 0.01
        }
    ]
}
```

The preliminary specification is the following:

- "type" should be "SunJSON";
- "time" is optional (not used currently);
- "geometry" is an array and its elements are defined as:
  - "type" (of geometry) can be:
    - ⋆ "ellipse" - should have exactly three coordinates specifying the ellipse parametrization: $\mathbf{x} = \mathbf{c} + \mathbf{u} \times \cos(t) + \mathbf{v} \times \sin(t)$, t=[0,2π];
    - ⋆ "line" - line segments, at least two coordinates;

[1] https://geojson.org

    * “point” - at least one coordinate.
- “coordinates” is an array of float triplets: [radius, Carrington longitude, Carrington latitude], expressed in [Rsun, degree, degree];
- “colors” is an array of integer quadruplets [R, G, B, A], 0-255; if array shorter than “coordinates” the last quadruplet is repeated (optional, green is used if missing); for ellipse only first color matters;
- “thickness” is a JHV internal value (subject to change).

6.1.7 Connecting remote-sensing and in-situ data (WP30500)

The JHV client will be updated to integrate in-situ data with the remote sensing data.

6.2 Task 4: Perform updates on the JHV server-side

6.2.1 Provide access to SOHO data from ROB server (WP30400)

This was implemented by fully mirroring the existing SOHO data from the GSFC server. Setting setting up a periodic synchronization of new SOHO data is pending stabilization of the GSFC server.

6.2.2 Improve server-side JPEG2000 data processing (WP30400)

This will be implemented by reviewing the JPX merging capabilities of the `hvJP2K` package and setting up the ROB server to use the JP2 headers stored in the database.

6.2.3 Investigate alternative setup for the Helioviewer image services based on IPFS (https://ipfs.io) (WP30400)

This investigation will follow the guidelines set by the corresponding requirement.

6.2.4 Provide access to and visualization of additional data sources (Hinode/EIS, IRIS, RHESSI) (WP30800)

Appropriate datasets with validated metadata were prepared on the server-side and loaded via SAMP in JHV for visualization. Those datasets are essential for Coronal Rain research.

6.2.5 Include access to catalogues via the VO (WP30800)

Server-side generation of image request files corresponding to event time spans (RHESSI flares, coronal rain events) and relevant datasets. Files loaded via SAMP (e.g., http://swhv.oma.be/test/samp/).

6.2.6 Providing access to science quality data (WP30800)

A library that bundles functions to connect between IDL/Python and JHV via SAMP protocol was created as separate project samp4jhv on GitHub. It contains examples to export images from JHV to SAMP and access them then in an environment commonly used for data analysis such as IDL or Python, as well as examples to load data such as RHESSI flares from Python to JHV. To reduce the list of dependencies needed to use this functionality, efforts were made to include as much of these functionalities directly in standard science packages such as SunPy. An official side-package of SunPy is sunkit-instruments, to which we added new routines for extracting image maps from RHESSI data cubes.

6.2.7 Interaction and interoperability with ESAC archives (SOAR, ADQL/TAP) (WP30800)

A DaCHS service providing ADQL/TAP access to RHESSI flares and IRIS observations was set up at https://tap.cs.technik.fhnw.ch/.

6.2.8 Visualization of new datasets (WP30900)

The JHV client was updated for the visualization of the new datasets of WP30800 and the integration of the new data access methods of WP30800 into the client.

6.3 Task 5: Use case 1: Limb-occulted X-ray flares in preparation for the era of Solar Orbiter

6.3.1 Add new annotation types (WP30700)

The circle and cross annotation types were updated with measurements of their diameter and height above solar sphere, respectively, which are displayed in the status panel. A new loop annotation was also introduced. It displays a measurement of its top height above the solar sphere in the status panel. The loop annotation can also visualize a line between the viewpoint and its top (line-of-sight).

6.3.2 Add support for RHESSI maps (WP30600)

Archive to be filled at https://hesperia.gsfc.nasa.gov/rhessi_extras/flare_images/. Several energy bands, several reconstruction algorithms. Data cubes are sliced into 2D FITS maps and appropriate metadata is added. JPEG2000 files can be created with `fits2img`.

6.3.3 Create EPN-TAP list for limb-occulted flares (WP30600)

The list is based on the existing RHESSI event list. The information about the position of the flare is provided there, therefore we can identify candidates for "limb-occulted" flares if the position is outside the solar radius. Further, we use manually confirmed limb-occulted flares from science publications, where available. The EPN-TAP compatibility will be established by using a DaCHS server.

6.3.4 Python examples to send RHESSI maps to JHV (WP30600)

This allows to build a direct connection via SAMP from the list of limb-occulted flares to the respective visualizations available in JHV. Examples can be found on samp4jhv on GitHub.

6.4 Task 6: Use case 2: Implement functionality for Solar Orbiter science planning and science exploitation (WP30700)

This use case has been implemented and will be subject to further refinements. It is implemented by the possibility of setting of the viewpoint, the new `FOVLayer` which draws the FOVs of several instruments, including the remote-sensing instruments of Solar Orbiter, and the optional distortion of images according to solar differential rotation. The off-pointing of FOVs is implemented by direct input in arcseconds, which also enables the direct output of the numerical values.

6.5 Task 7: Use case 3: Science Exploitation: research on coronal rain (WP30600)

In this use case JHV is being used to evaluate an algorithm that identifies IRIS observations which contain coronal rain. This is done by cross-correlating it with SDO and check the visualized data for visible indicators of coronal rain.

6.5.1 Create EPN-TAP compatible list of coronal rain events

The list of coronal rain events will be created by manually checking candidates from an existing clustering analysis. The list will be made EPN-TAP compatible by using the same procedure as for RHESSI events (i.e., on https://tap.cs.technik.fhnw.ch/)

6.5.2 Python examples to send coronal rain event data to JHV to display the corresponding IRIS/EIS data

This allows to build a direct connection via SAMP from the list of limb-occulted flares to the respective visualizations available in JHV. A separate, specialized tool can be used to analyze the corresponding IRIS spectra. An example of sending IRIS/Hinode EIS to JHV is available on samp4jhv on GitHub.

6.5.3 Functionality to output space-time data-cubes

Examples to load space-time data-cubes from VSO into a Python/IDL data analysis environment via SAMP messages are available on samp4jhv on GitHub.

6.6 Task 8: Code maintenance, software release, testing, bug fixing and user support (WP31000)

This is an ongoing activity and has seen already significant progress at this phase of the CCN3 activity.

7 Traceability matrix

The following table allows to trace the status of requirements for the duration of the project.

A list of changes with significant impact for the users is maintained at https://github.com/Helioviewer-Project/JHelioviewer-SWHV/blob/master/changelog.md.

The Status can be:

- Retired : marked Out of scope at PDR or by a later decision
- Superseded : marked as superseded at PDR or by a later decision
- Not started
- Initiated
- Implemented : can still be augmented considering new information
- Ready for acceptance : considered fully implemented for the scope of this project

The notes contain the planned stage of the requirements readiness for review.

Table 7.1: Requirements status at 2021-11-26

Requirement	Priority	Status	Notes
SWHV-CCN3-T3-01	Essential	Ready for acceptance	QR: independent time setting
SWHV-CCN3-T3-02	Essential	Ready for acceptance	
SWHV-CCN3-T3-03	Essential	Ready for acceptance	QR; ongoing activity
SWHV-CCN3-T3-04	Essential	Ready for acceptance	CDF format is TBD
SWHV-CCN3-T3-05	Essential	Superseded	
SWHV-CCN3-T3-06	Essential	Ready for acceptance	
SWHV-CCN3-T4-01	Essential	Implemented	QR: periodic updates depending on GSFC server availability
SWHV-CCN3-T4-02	Essential	Initiated	AR
SWHV-CCN3-T4-03	Desirable	Superseded	
SWHV-CCN3-T4-04	Desirable	Not started	AR: if sufficient resources

Requirement	Priority	Status	Notes
SWHV-CCN3-T4-05	Essential	Ready for acceptance	JPEG2000 datasets not required at this stage
SWHV-CCN3-T4-06	Essential	Ready for acceptance	
SWHV-CCN3-T4-07	Essential	Ready for acceptance	
SWHV-CCN3-T4-08	Essential	Not started	AR: depending on SOAR TAP accessibility
SWHV-CCN3-T5-01	Essential	Ready for acceptance	Requirement adapted
SWHV-CCN3-T5-02	Essential	Ready for acceptance	
SWHV-CCN3-T5-03	Essential	Ready for acceptance	
SWHV-CCN3-T5-04	Essential	Ready for acceptance	
SWHV-CCN3-T5-05	Essential	Ready for acceptance	Requirement adapted
SWHV-CCN3-T6-01	Essential	Not started	QR: related to SWHV-CCN3-T3-01
SWHV-CCN3-T6-02	Essential	Ready for acceptance	
SWHV-CCN3-T6-03	Secondary	Ready for acceptance	
SWHV-CCN3-T6-04	Essential	Ready for acceptance	
SWHV-CCN3-T6-05	Essential	Ready for acceptance	Implemented with spinner boxes
SWHV-CCN3-T6-06	Desirable	Ready for acceptance	
SWHV-CCN3-T7-01	Essential	Ready for acceptance	
SWHV-CCN3-T7-02	Essential	Ready for acceptance	
SWHV-CCN3-T7-03	Essential	Ready for acceptance	QR
SWHV-CCN3-T8-01	Essential	Ready for acceptance	AR; ongoing activity
SWHV-HS-01-01	Essential	Ready for acceptance	Configurable number of lines not necessary given the possibility of selecting the control point
SWHV-HS-02-01	Essential	Initiated	AR: depending on the server availability of EUHFORIA datasets
SWHV-HS-02-02	Desirable	Superseded	

Requirement	**Priority**	**Status**	**Notes**
SWHV-HS-03-01	Essential	Ready for acceptance	SoloHI supported
SWHV-HS-03-02	Out of scope	Retired	
SWHV-HS-03-03	Desirable	Not started	AR: depending on the availability of the relevant datasets
SWHV-HS-04-01	Essential	Initiated	AR
SWHV-HS-04-02	Secondary	Initiated	AR
SWHV-HS-04-03	Essential	Ready for acceptance	QR
SWHV-HS-04-04	Essential	Implemented	AR: planetary transits
SWHV-IT-01-01	Essential	Ready for acceptance	See SWHV-CCN3-T6-04 & SWHV-CCN3-T6-05
SWHV-IT-01-02	Essential	Ready for acceptance	QR
SWHV-IT-01-03	Desirable	Superseded	
SWHV-IT-01-04	Desirable	Ready for acceptance	Implemented for the FOV layer
SWHV-IT-01-05	Essential	Ready for acceptance	See SWHV-HS-01-01; additional model visualization via Connection layer
SWHV-IT-01-06	Essential	Ready for acceptance	QR: implemented with data from http://connect-tool.irap.omp.eu
SWHV-IT-01-07	Desirable	Ready for acceptance	QR: implemented with data from http://connect-tool.irap.omp.eu
SWHV-IT-01-08	Secondary	Ready for acceptance	Implemented with the “relative longitude” feature
SWHV-IT-01-09	Out of scope	Retired	

Requirement	Priority	Status	Notes
SWHV-IT-01-10	Desirable	Retired	Retired 2021-09
SWHV-IT-02-01	Desirable	Not started	AR
SWHV-IT-02-02	Desirable	Not started	AR
SWHV-IO-01-01	Essential	Not started	AR: depending on accessibility via SOAR TAP
SWHV-IO-01-02	Secondary	Not started	AR: depending on accessibility via SOAR TAP
SWHV-OT-01-01	Essential	Ready for acceptance	
SWHV-OT-01-02	Essential	Ready for acceptance	
SWHV-OT-01-03	Secondary	Retired	Retired 2021-09
SWHV-CCN3-UR-01	Desirable	Not started	AR: very difficult in the current design
SWHV-CCN3-UR-02	Desirable	Ready for acceptance	
SWHV-CCN3-UR-03	Desirable	Ready for acceptance	
SWHV-CCN3-UR-04	Desirable	Not started	AR
SWHV-CCN3-UR-05	Desirable	Retired	Retired 2021-09
SWHV-CCN3-UR-06	Desirable	Not started	AR
SWHV-CCN3-UR-07	Desirable	Not started	AR
SWHV-CCN3-UR-08	Desirable	Not started	AR: very difficult and low priority
SWHV-CCN3-UR-09	Desirable	Implemented	SunJSON functionality, to be incrementally improved for AR
SWHV-CCN3-UR-10	Desirable	Ready for acceptance	UI transition will not be implemented

`id: 4f6e1210e91f51439712e07f6f2b996ff6a6336e`

8 Acronyms and Abbreviations

ADQL	Astronomical Data Query Language
API	Application Programming Interface
AR	Active Region
ASCII	American Standard Code for Information Interchange
CACTus	A software package for “Computer Aided CME Tracking”
CALLISTO	International network of solar radio spectrometers
CCN	Contract Change Notice
CDF	Common Data Format
CDR	Critical Design Review
CH	Coronal Hole
CME	Coronal Mass Ejection
COMESEP	Coronal Mass Ejections and Solar Energetic Particles: Forecasting the Space-Weather impact, an EU-FP7 Space project
CRD	Customer Requirements Document
CSV	Comma-separated values
DaCHS	GAVO’s Data Center Helper Suite
DD	Design Document
DEM	Differential Emission Measure
DSCOVR	NOAA Deep Space Climate Observatory
EDC	EUI Data Center
ESA	European Space Agency
EUHFORIA	EUropean Heliospheric FORecasting Information Asset
EUV	Extreme Ultraviolet
FFT	Fast Fourier Transform
FHNW	University of Applied Sciences North-Western Switzerland
FITS	Flexible Image Transport System
FOV	Field of View
FTP	File Transfer Protocol
FSI	EUI Full Sun Imager
GAVO	German Astrophysical Virtual Observatory
GOES	NOAA Geostationary Operational Environmental Satellite
GOES/SXI	The “Soft X-ray Imager” on-board GOES
GONG	Global Oscillation Network Group
GSFC	Goddard Space Flight Center
GUI	Graphical User Interface

HEK	Heliophysics Event Knowledgebase
HER	Heliophysics Event Registry
Hinode	JAXA solar mission
Hinode/EIS	“EUV imaging spectrometer” on-board Hinode
Hinode/SXT	“Soft X-ray Telescope” on-board Hinode
HRI	EUI High Resolution Imager
HTTP	HyperText Transfer Protocol
IAS	Institut d’Astrophysique Spatiale
IMF	Interplanetary Magnetic Field
IRIS	NASA Interface Region Imaging Spectrograph
IVOA	International Virtual Observatory Alliance
JAR	Java ARchive
JAXA	Japan Aerospace Exploration Agency
JDK	Java Development Kit
JHV	ESA JHelioviewer project
JNI	Java Native Interface
JPEG	Joint Photographic Experts Group
JPEG2000	Image compression standard from the Joint Photographic Experts Group
JPIP	JPEG2000 Interactive Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LMSAL	Lockheed Martin Solar & Astrophysics Laboratory, PI institute of SDO/AIA
MK4	Mark-IV K-coronameter
NASA	National Aeronautics and Space Administration
NGDC	National Geophysical Data Center
NLP	Natural Language Processing
NOAA	National Oceanic and Atmospheric Administration
NRH	Nançay RadioHeliograph
NSF	National Science Foundation
NSO	National Solar Observatory
P3SC	PROBA-3 Science Operations Center
PDR	Preliminary Design Review
PFSS	Potential Field Source Surface
PROBA	ESA Project for OnBoard Autonomy
PROBA2/LYRA	The “Large Yield RAdiometer” on-board PROBA2
PROBA2/SWAP	The “Sun Watcher using Active Pixel System detector and Image Processing”, EUV imager on-board PROBA2
PROBA-3/ASPIICS	“Association of Spacecraft for Polarimetric and Imaging Investigation of the Corona of the Sun”, the coronagraph on-board PROBA-3
PSP	NASA Parker Solar Probe
REST	Representational State Transfer
RHESSI	NASA “Reuven Ramaty High Energy Solar Spectroscopic Imager” mission
ROB	Royal Observatory of Belgium

ROB/USET	The “Uccle Solar Equatorial Table” located at ROB
SDO	NASA Solar Dynamics Observatory
SDO/AIA	The “Atmospheric Imaging Assembly” on-board SDO
SDO/EVE	The “EUV Variability Experiment” on-board SDO
SDO/HMI	The “Helioseismic and Magnetic Imager” on-board SDO
SOAR	Solar Orbiter Archive
SOHO	ESA/NASA Solar and Heliospheric Observatory
SOHO/EIT	The “Extreme ultraviolet Imaging Telescope” on-board SOHO
SOHO/LASCO	The “Large Angle and Spectrometric Coronagraph” on-board SOHO
SOHO/MDI	The “Michelson Doppler Imager” on-board SOHO
SOL	Solar Object Locator
SOLIS	Synoptic Optical Long-term Investigations of the Sun
SOLO	ESA Solar Orbiter mission
SOLO/EUI	The “Extreme Ultraviolet Imager” on-board Solar Orbiter
SOLO/METIS	The coronagraph on-board Solar Orbiter
SOLOHI	The “Heliospheric Imager” on-board Solar Orbiter
SOOP	Solar Orbiter Observing Plan
SOW	Statement of Work
SPoCA	The “Spatial Possibilistic Clustering Algorithm” software package that detects and tracks Active Regions and Coronal Holes
SRD	System Requirements Document
SSA	ESA Space Situational Awareness programme
SSA-SWE	Space Weather Element of SSA
SSW	Solar SoftWare, a IDL library for solar physics
STEREO	“Solar TErrestrial RElations Observatory”, the third mission in NASA Solar Terrestrial Probes program
STEREO/COR	The coronagraph on-board STEREO
STEREO/EUVI	The “Extreme Ultraviolet Imager” on-board STEREO
STEREO/HI	The “Heliospheric Imager” on-board STEREO
STEREO/SECCHI	The “Sun Earth Connection Coronal and Heliospheric Investigation” on-board STEREO
SWEK	Space Weather Event Knowledgebase
SWHV	Space Weather Helioviewer
SWPC	NOAA Space Weather Prediction Center
TAP	Table Access Protocol
TRACE	NASA Transition Region And Coronal Explorer
UV	Ultraviolet
VOEvent	Virtual Observatory Event
VSM	Vector Spectromagnetograph
VSO	Virtual Solar Observatory
WP	Work Package

`id: bc1f9abc2f16c0600a8bb6decb56d0399641b606`