Webjpip - Handover

Summary

This page is to act as an entry-point for further investigation and development of a web visualisation tool for astronomy images that uses the JPEG2000 standard (ISO/IEC 15444) and Webjpip, an open-source web client. It also serves as the final report and conclusions of my (Anthony Carbone) 2020/21 summer project with supervisor Dr Slava Kitaeff.

During the 10 week summer project I investigated Webjpip's potential for AusSRC use in remote web visualisation of ASKAP (and future SKA) images. I do wish I could have made an appreciable contribution to the software but unfortunately lacked the web development experience to do so. Instead I focused on exploring all the avenues of existing technologies that could be of benefit to someone wishing to adopt Webjpip for use with spectral data cubes.

Definitions

channel: Associated with a single target (image), the channel identifies a single request queue for the server.

session: Enables stateful serving of a client, ie the server maintains a cache model of the client session. A session can have multiple channels for flexibility. At the start of a session, the client can communicate what databins they have in their cache so the server doesn't send these back.

SIDC: Spectral Imaging Data Cube.

ROI: Region of Interest. A region of the image that may be encoded/decoded in greater quality than surrounding regions. See Bradley, Andrew & Stentiford, Fred. (2002). JPEG2000 and region of interest coding.

IDF: Shorthand for the image-decoder-framework.js library also written by MaMazav for viewing decoded images in Leaflet/Cesium. See the Github repository and the documentation.

JPEG2000 and JPIP

The JPEG2000 standard (ISO/IEC 15444) comprises of 16 parts. The standard defines a lossless and lossy image compression method file format (Part 1), flexible extension features (Part 2), the JPIP interactive protocol (Part 9), and more.

There already exists some good resources that summarise these parts of the standard:

- A really good summary of the core JPEG2000 encoding scheme: C. Christopoulos, A. Skodras and T. Ebrahimi, "The JPEG2000 still image coding system: an overview,"
- A very brief Medium article describing JPIP: JPIP Protocol for dummies
- A more in-depth summary of JPIP: David S. Taubman and Robert Prandolini "Architecture, philosophy, and performance of JPIP: internet protocol standard for JPEG2000"
- An early (and out of date) JPIP summary: Proposal and Implementation of JPIP (Jpeg2000 Internet Protocol) in Kakadu v3.3

and of course the international standards themselves which are definitely recommended for reference. Check if your organisation or tertiary institution offers access to these documents before buying them.

The rest of this page assumes reasonable knowledge of JPEG2000, the extensions defined in Part 2, and JPIP (Part 9).

Use In Astronomy (SIDCs)

The features of JPEG2000 offer great benefits for astronomy images, particularly in random-access, ROI, compression efficiency, and flexibility in decoding. Slava Kitaeff has investigated applying JPEG2000 to astronomy, in Astronomical Imagery: Considerations For a Contemporary Approach with JPEG2000 and the effects of JPEG2000's lossy compression on radio astronomy images, The impact of JPEG2000 lossy compression on the scientific quality of radio astronomy imagery.

He was also involved in developing SkuareView, a prototype JPIP client for spectral data cubes. See SkuareView: Client-Server Framework for Accessing Extremely Large Radio Astronomy Image Data. The Github repository with SkuareView's source code is private to ICRAR.

Webjpip

The Github repository webjpip.js was the focus of my summer project, essentially gauging its potential for application for SIDCs, and making recommendations to AusSRC for further investigation and development.

Webjpip can be described as a pure web JPIP client written in vanilla Javascript. It was created by MaMazav in 2015, and contributed to by two others. It however has received little light despite its novelty, an unfortunate consequence of which is little and outdated documentation (see the W

ebjpip documentation, and outdated documentation.doc), few examples, and sparse maintenance. You can see my rough commenting on my fork if it helps at all.

Webjpip is ultimately intended for in-browser viewing of JPEG2000 images. It also supports progressive display of images (a critical benefit of using JPEG2000).

Architecture

Webjpip is built upon image-decoder-framework.js (IDF herein), a library created also by MaMazav (see its documentation site). IDF is described as "A framework for heavy decoding of images, including ability to view in Leaflet and Cesium". Essentially it serves an image supplied by a decoder framework (Webjpip in our instance) as a layer for Leaflet/Cesium.

IDF also uses yet another MaMazav library dependency-workers.js (see the documentation) which manages the web workers. This again uses async-proxy.js (see the documentation).

Webjpip in essence does 3 things (relative to the root of Webjpip);

1. sends the appropriate JPIP requests to the server and manages the session and channel (`src/protocol/`),
2. parses the JPIP responses and saves the image databins in local variables (`src/image-structures/`), and
3. decodes the image structure for display in Leaflet/Cesium (`src/api/`).

The decoder uses Mozilla's open-source pdf.js's jpx.js decoder. Webjpip's `src/api/pdfjs-jpx-pixels-decoder.js` then forms a pixel array from jpx.js's `JpxImage` to supply IDF an image, displaying it in Leaflet/Cesium.

Build

Webjpip was built using Webpack 3.10 (see Webpack 3.x documentation). Beware the bundler is now on Webpack 5.x, it may be worthwhile to update to Webpack 5.x.

It also has extensive functional mock tests that all pass. It would be beneficial to continue this convention on improving the library with similar testing.

API

Webjpip's `JpipImage` is the main class that IDF interfaces with. `JpipImage`'s only methods relate to progressiveness, that is how quality layers of the JPEG2000 image are loaded into the viewer. These are documented in the Webjpip Documentation.

Leaflet vs Cesium

IDF enables viewing through a simple HTML Canvas, the Cesium viewer, and Leaflet. I chose to use Leaflet throughout my investigation because its widely used, has a simple interface, has great support for plugins, and has good examples. Cesium may be harder to use and customise for its complexity, it seems to be now more oriented towards 3D mapping applications, but may be worth investigating.

It must be noted that Webjpip uses Leaflet 0.7.7 which is a legacy version (it is now on Leaflet 1.7.1). After 0.7.7, Leaflet underwent a significant rework, particularly with the layer API that IDF uses to display its images. It may be worthwhile to update Webjpip to function with the new Leaflet. Leaflet 1.x also has better support for plugins, useful for exploration of image cubes (see recommendations).

Server-Side

I used Kakadu's `kdu_server` to serve JPEG2000 images. As good as this application is, it has some unfixed bug that after serving some clients, it refuses to open a channel and serves them statelessly.

I investigated OpenJPEG's OpenJPIP application but found it difficult compiling on my machine and to work with Apache. It does however evidently work, and would benefit from investigation.

Kakadu vs OpenJPIP

I encourage the use of (and most likely contribution to) OpenJPIP over using Kakadu as its open-source and not proprietary like Kakadu. Due to licensing, the Kakadu is strictly not to be distributed to non license holders, and so inhibits its potential. OpenJPIP may also be considered to have greater certainty over long-term maintenance given it is open-source, whilst Kakadu is primarily maintained by one David Taubman. This may however require developing the less-mature OpenJPIP, though would be very beneficial to the open-source community.

CORS

It must be noted, that as the client is making requests from a web-browser to the server which is most likely to be on a different origin, Cross Origin Resource Sharing (CORS) must be enabled (see this MDN page).

MaMazav in Webjpip's README hinted that `kdu_server` could be easily edited to include the required HTTP headers to enable CORS, however I failed in my attempts to do so. I instead opted for a simple node.js proxy using the `http-proxy` module. The headers that were sufficient to add to the responses were:

`Access-Control-Allow-Origin: *`, and

`Access-Control-Expose-Headers: JPIP-tid, JPIP-cnew`.

Ideally, a proxy should not have to be used and the JPIP server includes all the required headers in its response.

The simple node.js proxy I ran.

```
// Proxy to enable CORS for webjpip
// Set target, variables below include a local kdu_server and public
kdu_server
// Send webjpip requests to http://127.0.0.1:<proxyPort>/
// Responses from server will come with added headers for CORS

const http = require('http');
const httpProxy = require('http-proxy');

var target = 'http://127.0.0.1:8080'; // Use any target here

var proxyPort = 8081;

var options = {
  target: target,
  preserveHeaderKeyCase: true // REQ'D! Do not change the letter case
of the response
}

// Create server of proxy instance with target
console.log('Starting CORS Proxy...')
console.log(`Listening on http://127.0.0.1:${proxyPort}`);
console.log(`With target ${options.target}`);
var proxy = httpProxy.createProxyServer(options).listen(proxyPort);

// On proxyRes event, add headers
proxy.on('proxyRes', (proxyRes, req, res) => {
  res.setHeader("Access-Control-Allow-Origin", "*");
  res.setHeader("Access-Control-Expose-Headers", "JPIP-tid, JPIP-
cnew);
});

// Console log errors
proxy.on('error', (err, req, res) => {
  console.log('\n -- ' + `${err}` + '  -- \n');
  res.writeHead(500, {'Content-Type': 'text/plain'});
  console.log('Proxied response\n', JSON. stringify(res.headers,
true, 2));
})
```

Recommendations

In all, I make the following recommendations:

Investigate out of browser caching.

As is, all downloaded image data (usually runs into the tens of MBs, but may be more) are stored in the local Javascript variables during runtime. This leads to all downloaded image data being lost on refreshing the page, and becomes essentially cache-less between sessions. This may be acceptable for small case uses, but would be unfortunate for frequent users as its not fully taking advantage of the JPIP caching feature.

A Stack Overflow question seemingly regarding Webjpip sounds like this with no compelling answers. Ideally, the browser would store a local cache on the system rather than in-browser as locally the user has full-control whether they wish to retain or discard it. In-browser it is up to the browser how and if it stores this information. I'm not sure if IndexedDB performs such actions.

Update Webjpip dependencies.

Webjpip has a number of dependencies, particularly Webpack 3.10.0, jshint 2.9.5, and some babel modules. These are all quite out of date, particularly Webpack. These may be worthwhile to update.

Webjpip's package.json devDependencies section.

```
"devDependencies": {
  "babel-loader": "^7.1.2",
  "babel-minify-webpack-plugin": "^0.2.0",
  "babel-preset-es2015": "^6.24.1",
  "jshint": "^2.9.5",
  "jshint-loader": "^0.8.4",
  "npm-run-all": "^4.1.2",
  "webpack": "^3.10.0",
  "webpack-dev-server": "^2.11.5"
}
```

Update Leaflet compatibility.

Currently, only the legacy version 0.7.7 Leaflet is supported with IDF (remember IDF takes the image from Webjpip and creates a layer for Leaflet). Leaflet is now on version 1.7.1. From 0.7.7 to 1.0 there was a number of significant API changes, particularly with tile layers, that essentially break IDF's functionality.

Updating Webjpip to match Leaflet's 1.x Layer API would reap the benefits of better support for building plugins, which I recommend later.

Develop Mozilla's jpx.js decoder for SIDC use.

As is, jpx.js has minimal support for jpx images with many components (that is not greyscale, RGB, or RGBa). As it is this decoder that Webjpip uses, jpx.js must support this. I suggest that jpx.js (pdf.js) be built an API that will permit viewing of many component jpx images, to support SIDC's. This would be used by the next recommendation.

Leaflet/Webjpip Plugin.

Leaflet 1.x has greater potential and neater plugins. A plugin could be created to change the 'component' of the image to be viewed in Leaflet.

This would need to interface with the aforementioned pdf.js API to decode single components.

Use OpenJPIP as server.

As Kakadu's proprietary licensing prohibits its wide-spread distribution, I encourage OpenJPIP to be investigated to act as the sole JPIP server application. Personally, I found it hard compiling and ran out of time in the project to demonstrate a simple demo with it, but gather it does indeed achieve its requirements.

I assume that the CORS issue is best-dealt at the server itself. Thus ideally OpenJPIP would include all the required CORS headers in the HTTP messages by default.

Enhance Leaflet/Webjpip interface for scientific use.

This recommendation is to achieve an appreciable scientific use of the Webjpip, that is performing early analysis. SkuareView for instance had several tools including a spectrum analysis tool. Other reference softwares like TopCat are good examples of what tools could be included.

These tools could be built as a GUI in the Leaflet plugin, and have it interface with the decoded image from jpx.js.

Other Learnings

These are some miscellaneous things I learnt playing around with the software that didn't fit in the above sections.

- SkuareView demonstrated much 'cleaner' session and channel management that Webjpip. This likely is due to SkuareView being based on Kakadu code, like that of `kdu_server` that I was using, and so all the small bugs had been dealt with. Webjpip would likely require modification to perform better in session and channel management.
- kdu_server would crash with segmentation faults with many images when I was serving them locally on my machine. This included images I compressed using `kdu_compress` and the Dingo images I was using (I tried two 12GB Dingo images locally). I tried different compression settings like those recommended in Kakadu's `Usage_Examples.txt` to no avail.