

# ***Simple Image Retrieval Interface Concepts and Issues***

*D. Tody, July 2002*

## **Scope and Requirements**

- Like the simple cone search, simple image retrieval is considered a prototype, intended to demonstrate basic capabilities early in the project while we develop the concepts and technology required for a more comprehensive VO framework.
- A primary goal is to support the first-year science prototypes. These require a means to query remote data collections and retrieve image data in a region of interest.
- The image retrieval service should be an open standard, requiring no centralized control (other than providing some standard service registries), allowing anyone to publish an image service, and providing a standard interface to such services.
- It should be as easy as possible for a service provider to publish an existing image service without having to make changes to their service.
- While keeping things simple is a primary goal, the image retrieval facility has to be sophisticated enough to deal with real-world image services. In particular, an automated image retrieval service may require substantial time (e.g. minutes) to retrieve image data from near on-line storage, compute a large mosaic, etc., and it should be possible to make use of such services.
- Security and authentication issues, including limiting access to proprietary data, will not be addressed initially. All data and services are initially assumed to be public.
- Data model issues will not be addressed initially, other than providing minimal guidelines for the data returned by the service (e.g., a valid WCS in some standard format is required). Image data will be returned as FITS or in a graphics format such as JPEG. Embedding image data directly in a VOTable is not possible without further development of data models.

## **Implementation Technologies**

To implement a simple image query and retrieval service, should we use a simple URL-based interface, or WSDL/SOAP?

- The simplest "web services" provide a conventional URL for the request, returning data in any legal MIME type (usually XML or `image/<type>`) as the response. The simple cone search is an example of this, as are most existing stand-alone image cutout services.

The advantage of this approach is that it is simple and works with a simple http-based interface. We don't have to rush to use Web Service technology and toolkits.

- The emerging Web Services technology (SOAP/WSDL) provides a means to implement a real distributed-computing interface with methods, argument serialization, etc., which can be dynamically bound to various languages or runtime environments. Data encoding is normally done using SOAP transparently to both the client and the service. Data "as sent over the wire" is normally encapsulated within a special XML-based format (SOAP) for both requests and responses.

The advantage of this approach is that it is more implementation and language independent, and automates the mechanics of getting two programs to talk to each other over the network. However, both the client and server have to use WS infrastructure (typically a framework/toolkit of some sort) for this approach to work, and data normally needs to be encapsulated in SOAP to gain all the advantages of web services.

- It is possible to define a service in the abstract and represent it using a simple URL-based interface, via SOAP/WSDL, via other protocols. In one case the arguments are passed in the URL (e.g., `<base-url>?param=val%param=val%...`), while in the other they are serialized and encoded via SOAP and passed over the wire as a block of XML text. Logically the interface is much the same in both cases.

It should be possible to define a service interface logically and implement both the client and server code largely independently of the execution protocol used (e.g., URL/HTTP GET or SOAP). Hence, given a good logical interface design, we can start with a simple URL implementation and later implement much the same interface as a full-up web service using WSDL/SOAP (as for example, OpenGIS has already demonstrated).

## Interface Summary for Simple Image Retrieval

We would like to define an interface which is simple to implement and use for "simple" image services (those which provide synchronous access to to a single image cutout), but capable of supporting more complex services which may need to stage or batch data, possibly in parallel.

It would be nice to not have to deal with two different interfaces to handle these two cases; ideally the most important methods should be the same in both cases. A possible solution is to provide a single interface which uses the same query and image retrieval methods for both types of services, augmenting the interface with optional methods which are only required to optimize access to asynchronous services. For simple services these methods can be omitted or stubbed out by the server.

For example:

```

vt = queryImageRegion (region[, format, band, size, ...])

df = getImage (acref)

accessid = accessImages (acrefs[, client-addr, ...])
acrefs = getImageQueue (accessid)
flushImages (accessid)

```

Here we represent the Simple Image Retrieval image service symbolically; it could be implemented using URLs (HTTP GET), SOAP/WSDL, or any other execution framework and protocol. Here, "region" refers to a region on the sky such as a cone or rectangle (or more complex regions - see below), "vt" refers to a VOTable, "df" refers to a data file of some sort, e.g., image/FITS or image/JPEG, and "acref" refers to an "access reference" for an individual image datum. Note that the acref (which is probably a URL for a URL-based service) can be used to hide all the service-dependent access details for a particular service.

A simple cutout service would require only queryImageRegion (which returns a VOTable listing all image data intersecting the given region) and getImage (which gets a single image reference). A more complex asynchronous service, requiring that data be staged or computed, would use the second block of three methods shown above to asynchronously request that data be staged or otherwise generated (accessImages), to query the status of such a request during execution (getImageQueue), or to abort the entire operation or free staged data after retrieval when the data is no longer needed (flushImages). Both types of services would use the same queryImageRegion and getImage methods.

## Example 1: Simple synchronous image retrieval

The client (after doing service discovery using the service registry) queries a single registered image service to find all the image data available for a single region:

```
vt = queryImageRegion (region[, format, band, ...])
```

The client parses the returned VOTable and decides which image data, if any, it wishes to retrieve. Zero or more getImage invocations are then used to fetch the image data:

```
df = getImage (acref1)

df = getImage (acref2)
df = getImage (acref3)

```

[etc]

Each getImage request will block until the data (of type image/<imtype>) is returned to the client. If a timeout or error occurs, a status message is returned. If the timeout indicates that the data exists but could not be returned in the timeout interval, a subsequent call can be made to get the data. This provides a crude mechanism for dealing with asynchronous services using only simple calls.

If only high level image metadata is required, e.g., the image footprint, then only the `queryImageRegion` method may be needed.

## Example 2: Asynchronous or Bulk Image Retrieval

The client (after doing service discovery using the service registry) queries a single registered image service to find all the image data available for a single region:

```
vt = queryImageRegion (region[, format, band, ...])
```

The client parses the returned `VOTable` and decides which image data, if any, it wishes to retrieve. At this point it is possible to "poll" the service to fetch data with `getImage` as above, but more sophisticated, scalable web applications can be built using the asynchronous services.

Having analyzed the `VOTable` and selected the images to retrieve, the client uses the `accessImages` method to "access" the images that it wants to retrieve. This method requests that the server generate the referenced images and stage them for later retrieval by the client.

```
accessid = accessImages (acrefs[, client-addr, ...])
```

Each `accessImages` call takes as input a list of `acrefs` (access references), as returned by `queryImageRegion`, specifying the images to be staged. The method returns as soon as the service has accepted the request, but before any image data has actually been staged. An optional `client-addr` may be passed in by the client specifying a socket to receive messages from the server as image data becomes available. The service returns an "accessid" to uniquely identify each `accessImages` request to the service.

Not all client applications will be able to use messaging since the client may disconnect from the service after initiating a request. In such cases the `getImageQueue` method may be used to query the image service to find out which images have been staged and are ready for download. A list of all the images (`acrefs`) in the original request is returned, specifying the status of each image in the list. Like `accessImages`, this method returns immediately.

```
acrefs = getImageQueue (accessid)
```

Finally the client makes zero or more `getImage` calls to retrieve each "image" given its `acref` (an "image" may be a single image such as an image/FITS or an image/JPEG, or possibly something more complex such as an image/FITSMEF for multiband image cutouts).

```
df = getImage (acref1)
```

```
df = getImage (acref2)
```

```
df = getImage (acref3)
```

[etc]

Normally the server will delete staged image data after an interval determined by the server. If a client knows that it will never need to access a particular acref again, it can call `flushImages` to flush an acref or list of acrefs (flush means release any temporary storage occupied by the referenced image data). Optionally, an argument to `getImage` can be used to tell the server it is ok to flush the referenced data if the `getImage` returns successfully. The `flushImages` method can also be used to abort a prior `accessImages` request and free all server resources associated with the request.

## Future Interface Extensions

Extensions to the above interface can easily be made to batch image requests further, as may be needed to optimize large requests (e.g., permit efficient parallel execution). For example, a

```
vt = queryImageRegions (regions[, format, band, ...])
```

method could be added to query a list of regions on a single image server, e.g., for cases where many image cutouts need to be retrieved from a single data collection.

Another extension would be to replace "region" (specified as a simple box) with a more complex region, using some VO-standard region specification. An example where a complex region might be needed would be a query to obtain a list of all available data for a large irregular region such as the LMC, M31, etc.; multiple atlas images or image cutouts might be needed to return all the data for such a large region.

The basic image retrieval mechanism here is `getImage`, which intentionally returns a modest amount of data - the data corresponding to one acref, e.g., a single image cutout. This is done to make the interface 1) simple, 2) flexible, and 3) responsive. It is not clear apriori if batching data transfers will necessarily result in greater retrieval efficiency.

A case where this could be important might be when transferring many small image cutouts using HTTP. In this case one could either add a new "getImages" method with support for a new multi-image datafile type, e.g., gzipped-tar or VOTable with embedded image data, or possibly one could merely add support for batch transfers to `getImage` as a new datafile MIME type.

## Issues

- Unlike the cone search, which for a large catalog might return many object references in response to a query on a small region, an image query for a small region on a single data collection might return only a single image reference. If image services for individual data collections were separately registered, many such queries could be needed to implement data discovery for a region. Hence, unlike the cone search we might want to register image services which can query and access all (or selected large subsets of) the data collections at a site.

- If an image service can return data from multiple collections then the characteristics of different image references might vary from one acref to the next within the same query response VOTable. For example, one data collection might require staging while another is online for immediate access. One might generate cutouts of any size, while another might return fixed size atlas images.
- An image query on a large region might return multiple image references for that one region. For example, the region might be large and the service might have an upper limit on the size of an individual image cutout (as opposed to generating a mosaic of the large region requested), or the image service might return fixed-size atlas images, with many such images falling within the region requested. It is essential that the query method return "footprint" image metadata to allow the web application to decide what to do with the referenced images satisfying the query.
- It is probably best to deal with all image types with the same interface (that is, both science and graphics images, e.g., FITS and JPEG/PNG/GIF etc.). For example, image metadata, such as the image footprint, is useful for graphics images as well as science images, as are the image size and scale parameters in requests. The issue is whether to specify the image format in the query or in the getImage and accessImages methods. Perhaps this should be left up to the service, which can register a single service or multiple services to deal with both types of data.
- The message format for accessImages (for messages sent to the client as data becomes available on the server) is not clear. Probably this is a SOAP-encoded message for a conventional WSDL/SOAP web service. The server should probably broadcast such messages to all subscribing clients, without requiring any acknowledgment, and ignoring any errors.

## Methods

[TBA] We need to add a specification for the detailed arguments and semantics for each method. In particular, how is the region specified, what arguments does a method such as queryImageRegion accept, what fields are included in the returned VOTable, what are the error returns, what are the datafile types supported by getImage, and so forth.

## Implementations

[TBA] Specify URL-based implementation, which uses HTTP GET arguments to pass method arguments. Specify Web Services implementation, which uses WSDL/SOAP to pass arguments and (most) data back and forth between the client app and Web service.