*I*nternational

*V*irtual

*O*bservatory

*A*lliance

Homogeneous Access to Tabular Data

IVOA Note 01-May-2007

Authors:

Aurélien Stébé, Kona Andrews, Guy Rixon, Iñaki Ortiz

Abstract

This document describes a homogenized Protocol for accessing tabular data in the Virtual Observatory realm. This note only describes the Protocol to access the data, and not the query language nor any service-specific operations. These are described in other documents (ADQL, a Xmatch-like service spec, VOSI …).

Status of This Document

This is a Note. There are no prior released versions of this document.

Contents

1 Introduction

During the evolution of the IVOA and its activities, it has become clear that a number of different technology layers are involved in the general question of data access within the VO. The decision was taken to create a technical experts group to deal with the disentangling of these different layers. Such a group was created (VOQL Technical Experts Group, VOQL-TEG) on 01-Sep-2006.

The agreed structure for the different aspects of data access was deemed to contain three basic layers:

- The Query Language (Astronomical Data Query Language, ADQL)
- The Access Protocol (Table Access Protocol, TAP)
- The Services (e.g. a Crossmatch service)

This document addresses exclusively the Table Access Protocol layer.

2 Input Query Formats

In this protocol specification, two input query methods are defined. The first is intended to be easy to implement and to use, whereas the second is intended to give complete access to the dataset. The second method provides both synchronous and asynchronous access modes, the latter to support long-running queries. Either or both of these methods may be implemented, and asynchronous support is optional.

As may be seen below, these two methods do not individually define their output formats or their error handling. These aspects of the protocol are common to both query methods, and are defined in subsequent sections.

2.1 Simple Access Query

The Simple Access Query method is intended to be easy to implement both for the server and the client. It gives access to a dataset using simple Key-Value Pairs to filter the data to be returned, and allows only minimal control over the quantity of data (rows / columns) to be returned.

This access method presents the dataset as a flat, single table structure and effectively hides the dataset’s inner structure. Hence, usually only a subset of the data may be accessed and clients/users quickly reach the limits of the interface.

Nevertheless, it is simple to implement, and may be very suitable for publishing small and simple datasets. It can also provide immediate (if basic) access to more complex datasets while a more complete interface is being implemented.

2.1.1 Query method format

The Simple Access Query method is invoked via a standard **HTTP/1.1 GET** request to a URL endpoint. The URL endpoint takes the following general form:

```
http://service.endpoint.saq{?,&}PARAM=value[&…]
```

<u>The different elements constituting the URL endpoint are:</u>

- The HTTP protocol designation => `http://`
- The specific service endpoint => `service.endpoint.saq`
- The endpoint/parameters separator => `{?,&}`
- A list of query parameters formed by:
  - A parameter name => `PARAM`
  - A parameter name/value equal sign separator
  - A parameter value => `value`
  - A parameter list ampersand sign separator

Note that the specific service endpoint may include login information (e.g. “`user:pass@`”), port number (e.g. “`:8080`”) and an arbitrarily long path. Anchor pointers (e.g. “`#anchor`”) may be present in the URL endpoint, but they have no specific meaning in this protocol specification for the service.

The endpoint/parameters separator may be either a question mark or an ampersand depending on the presence or not of a question mark in the specific service endpoint.

Parameter names should avoid using unsafe characters, but if they do, they must be URL encoded. Parameter values must be correctly URL encoded. The comma and the slash characters have special meaning in this query method. They must be URL encoded to escape that meaning.

<u>The HTTP request is then:</u>

```
GET /path{?,&}PARAM=value[&…] HTTP/1.1
Host: service.endpoint.url
```

If no parameters are provided, the service may return all of its data (within its maximum returned rows limit, if this exists) and providing its default set of columns. The service may also return some standard introduction document instead. This allows the user to be presented with some documentation and/or an HTML web form when first accessing the service using a browser for example.

If one or more parameters are provided, they shall act on the data as a successive set of filters. The order in which the filters will be applied is undefined; it does not have to match the order in which the filter parameters are specified in the URL endpoint. Only data that satisfies all of the supported filters shall be returned.

If a parameter is unsupported by the service, it may be reported as an error or silently ignored. Parameters noted as compulsory in the protocol specification shall not be silently ignored; failing to support any of them will render the service non-compliant with the specification.

Parameter names are usually defined and written in uppercase, but they shall be treated in a case-insensitive manner by the service. A good habit followed by most services is to write protocol defined parameters in uppercase and service-specific ones in lowercase.

The parameters should not modify the output format of the service (see point 4 for this); they may only limit the quantity of data (how many rows), the type of data (specific values filtered) and the quantity of information (how many columns).

The service may also support this query method using the HTTP POST verb, but this should not be considered standard behaviour by clients. The other HTTP verbs (PUT, DELETE, OPTIONS, HEAD …) are undefined for this query method.

2.1.2 Generic PARAM types

Here we define the generic PARAM types that can be used to filter the data. These types describe *families* of possible parameters, where service implementers define the actual supported parameter names; this should allow server and client implementers to write generic code for handling input parameters.

Additionally, we specify a few reserved PARAM names, for which additional logic might be needed. Services or protocols that expand on this specification may define additional PARAMs, but they should be based on the following generic types.

2.1.2.1 The single value type

This is the simplest PARAM type. The parameter name points to a single data type (one column) and it must be compared to the value specified using an equal sign operator. The input looks like this:

```
PARAM_NAME=value
```

The value can be either a numerical type or a character string type. Hence, for numerical types it would result in the following interpretation in an SQL-like language:

```
PARAM_column = value
```

… and for character string types:

```
PARAM_column = “value”
```

The definition of the parameter may specify that it shall be treated as an upper or lower limit. This is only valid if used with a numerical type value. Hence, for upper limit parameters the interpretation would be:

```
PARAM_column > value
```

… and for lower limit parameters:

```
PARAM_column < value
```

2.1.2.2 The list value type

The parameter name also points to a single data type (one column), but the value specified is actually a comma separated list of values. The input looks like this:

```
PARAM_NAME=value1,value2,value3
```

The value can be either a numerical type or a character string type. Hence, for numerical types it would result in the following interpretation in an SQL-like language:

```
PARAM_column IN (value1,value2,value3)
```

… and for character string types:

```
PARAM_column IN (“value1”,”value2”,”value3”)
```

For character string types, the list may begin or end with a comma, and/or have two consecutive commas. This shall be interpreted as the empty string value and must be included in the list of values.

For numerical types, if the list begins or ends with a comma, and/or has two consecutive commas, this shall not be interpreted as the zero value and may be reported as an error by the service or silently ignored.

2.1.2.3 The interval value type

Here the parameter name still points to a single data type (one column), but the value specified is actually a slash separated interval. The input looks like this:

```
PARAM_NAME=valueMIN/valueMAX
```

The value may only be a numerical type. The interval may be open at one end (`PARAM_NAME=/valueMAX` or `PARAM_NAME=valueMIN/`), but it has of course no meaning to open at both ends (this is equal to not filtering on this value). This kind of input (`PARAM_NAME=/`) may be reported as an error by the service or silently ignored.

Hence, the resulting interpretations in an SQL-like language might be, as appropriate:

```
PARAM_column BETWEEN valueMIN AND valueMAX
PARAM_column > valueMIN
PARAM_column < valueMAX
```

`PARAM_column > valueMIN` (open upper limit interval)
`PARAM_column < valueMAX` (open lower limit interval)

2.1.2.4 The interval PARAM type

In this case, the parameter name points to an interval of values (two columns). The definition of the parameter shall specify this, as no indication in the parameter name is given. Also, this parameter type may only be used with numerical type values.

If the value is of the single type, then the filter consist of checking the value is inside the interval. The resulting interpretation in an SQL-like language is:

```
PARAM_columnMIN < value AND PARAM_columnMAX > value
```

If the value is of the interval type, then the filter consists of checking the two intervals cover each other. The resulting interpretations might be, as appropriate:

```
PARAM_columnMIN < valueMAX AND PARAM_columnMAX > valueMIN
```

`PARAM_columnMAX > valueMIN` (open upper limit interval)
`PARAM_columnMIN < valueMAX` (open lower limit interval)

2.1.3 Reserved PARAM names

The following list of parameters is by no mean compulsory for services supporting the Simple Access Query method, but if they are needed (or rendered compulsory by a service or protocol definition that expands on this one), then they must comply with the following definitions.

These PARAM names are reserved and may not be used for other purposes; additionally, the functionality associated with any of these reserved PARAM names should not be provided using any other PARAM name.

2.1.3.1 The POS & SIZE parameters

TODO (see the definition in the SSAP specification for the moment)

2.1.3.2 The BAND parameter

TODO (see the definition in the SSAP specification for the moment)

2.1.3.3 The TIME parameter

TODO (see the definition in the SSAP specification for the moment)

2.1.3.4 The FORMAT parameter

TODO (see the definition in the SSAP specification for the moment)

Note: this parameter does not indicate the desired output format of the query results. It has the same meaning as in the SSAP specification (filtering on the dataset format). To avoid confusion, this document will use the OUTPUT parameter, as described below, to select the output format for the time being.

2.1.3.5 The VERB parameter

TODO (see the definition in the SSAP specification for the moment)

2.1.3.6 The TOP parameter

TODO (see the definition in the SSAP specification for the moment)

2.1.3.7 The TOKEN parameter

TODO (see the definition in the SSAP specification for the moment)

2.2 Complete Access Query

The Complete Access Query method is intended to give total access and control over the dataset to the client. It uses a full query language (defined in other specifications) to access the data, which offers much more control than the Simple Access Query method.

This access method makes the dataset's inner structure available to the client. Hence, any and/or all columns of the data may in principle be returned, but the clients/users become responsible for data-level considerations such as table joins, order-by constraints or output column selection.

2.2.1 Query method format

The Complete Access Query method is invoked via a standard **HTTP/1.1 POST** request to a URL endpoint. The URL endpoint takes the following general form:

```
http://service.endpoint.caq
```

The different elements constituting the URL endpoint are:

- The HTTP protocol designation => `http://`
- The specific service endpoint => `service.endpoint.caq`

Note that the specific service endpoint may include login information (e.g. "`user:pass@`"), port number (e.g. "`:8080`") and an arbitrary long path. Anchor pointers (e.g. "`#anchor`") may be present in the URL endpoint, but they have no specific meaning in this protocol specification for the service.

The actual parameter is passed in the message body in the following form:

```
queryType=queryString
```

<u>The different elements constituting the message body are:</u>

- The query type name value => `queryType`
- The parameter name/value equal sign separator
- The actual query string value => `queryString`

The query string must be URL encoded as it would be if sent from an HTML form with the "`enctype`" set to "`application/x-www-form-urlencoded`".

<u>The HTTP request is then:</u>

```
POST /path HTTP/1.1
Host: service.endpoint.url
… (additional headers) …
CRLF
queryType=queryString
```

If no parameters are provided, the service may return an error or empty output. The service may also return some standard introduction document instead. This allows the user to be presented with some documentation and/or an HTML web form when first accessing the service using a browser for example.

Note that it is an error, and it must be reported as such, to pass any other parameter in the message body than the ones defined hereafter or to specify more than one parameter.

The service may support any number of the following query types. Trying to make a query using an unsupported query type shall result in an error output. Parameter names are usually written as defined hereafter, but they shall be treated in a case-insensitive manner by the service.

The service may also support this query method using the HTTP GET verb, but this should not be considered standard behaviour by clients. Moreover, clients should be aware that problems may arise due to the URL string length limitations of some network equipments. The other HTTP verbs (PUT, DELETE, OPTIONS, HEAD …) are undefined for this query method.

Note: the assumption is being made here that the service provides read-only access to the data, so that queries can have no side-effects. It seems likely that the next version of ADQL will guaranty this. The absence of side-effects might not be guarantied for the directQuery method discussed below.

2.2.2 The nativeADQL method

This method takes an ADQL string-formatted parameter in the form:

```
nativeADQL=SELECT * FROM table WHERE …
```

The ADQL specification, defined by the VOQL-TEG, can be found here (ref. [1]).

2.2.3 The uTypeADQL method

This method takes an ADQL string-formatted parameter in the form:

```
uTypeADQL=SELECT * WHERE …
```

The ADQL specification, defined by the VOQL-TEG, can be found here (ref. [1]).

This method differs from the nativeADQL method in that the query is assumed to be written against a formal data model, rather than against the service's published table and column names. The service must make the mapping from the formal data model to its actual table and column names before performing the query.

2.2.4 The directQuery method

Note: Alex Szalay (member of the VOQL-TEG) proposed to add a directQuery method. This method would take a standard SQL parameter or even vendor specific definition of SQL. This method should be understood as a pass-through to the database and should be implemented and used with extreme caution. It would be, in any case, up to the service provider to implement and activate this method or not. Some other members of the VOQL-TEG expressed their concern regarding security issues by allowing such a method.

Such a method might additionally be useful for services wishing to expose non-ADQL querying capabilities for specialized services using non-SQL query languages (e.g. XQuery, SPARQL …).

2.3 Asynchronous Querying

By default, the “simple” and “complete” queries are both synchronous: the client makes a single HTTP request and receives in response the query results. These modes require the client and service to stay connected, without network interruption, HTTP time-out or client down-time, for the entire length of the query. Very-long-running queries work more reliably if the client submits the query, disconnects, and calls back later to fetch the results (asynchronous operation).

The Asynchronous Query method is invoked via a standard **HTTP/1.1 POST** request to an URL endpoint. This endpoint must be distinct from the endpoints providing the simple-access query and (default, synchronous) complete-access query. Any request that may be posted to the complete-access endpoint may be sent instead to the asynchronous-query endpoint. The request format, and especially the query language, are identical with the exception that the asynchronous-query endpoint supports an additional parameter DEST.

2.3.1 The DEST parameter

When a query is submitted asynchronously, the results of that query do not return directly to the client and instead must be stored somewhere on behalf of that client, for later synchronous retrieval.

The location where the data is to be stored is controlled by the DEST parameter, which may take one of two forms. The first form uses the fixed keyword LOCAL:

```
DEST=LOCAL
```

This form indicates that the data should be staged at the service, for later collection by the client (“local delivery”).

The second form allows a destination URL to be specified, so for example:

`DEST=http://my.server/~john/out.vot` (an HTTP URL)
`DEST=ftp://my.server/~john/out.vot` (an FTP URL)
`DEST=vos://my.server!vospace/john/out.vot` (a VOSpace URL)

This form indicates that the service should deliver the data to the specified remote URL, using an appropriate method ("remote delivery"). If the service is supplied with a destination URL that involves an unsupported protocol, it should be reported as an error. Ideally the supported protocols will be published in the service registration.

Support for both local and remote delivery is optional; however, at least one of the two delivery methods must be supported by the asynchronous query endpoint.

2.3.2 Job management & UWS

The web resource at the asynchronous-query endpoint is a Job List object as defined by the Universal Worker Service (UWS) standard; this document uses v0.3 of the UWS standard (ref. [7]). Therefore, successfully posting a query here creates a UWS job. That job is represented as a set of separate web-resources following the UWS standard. The client may poll these new resources to monitor and control the execution of the query.

When the query completes, it produces a table of results exactly as if the query had been executed at the complete-access-query endpoint. The table of results is stored on the server and is made available for downloading as a web resource. The table is the formal, UWS result named "table" and this allows the client to discover its URI via the UWS interface. The service may store the table inside its web application, or it may store it on an external web server or FTP server.

UWS allows a job to be destroyed either by command of the client or by timing out. Destroying a job running a query involves, in addition to the normal UWS requirements, aborting the query on the underlying DBMS.

All other aspects of job control work here as detailed in the UWS standard.

3 Output Result Formats

Hereafter, we briefly discuss a new idea for output result formats handling. The advantage of this solution is that the mechanism remains consistent across different input query methods. The disadvantage is that the output format requested must be a MIME type format, and that the method itself is not universally applicable (for example, it may not work in the asynchronous case).
In addition to this new method, a more traditional approach is supported using an OUTPUT parameter to control the output format selection. NOTE: this OUTPUT parameter is not equivalent to the FORMAT parameter used in S*AP protocols.

A similar comment could be made in this specification concerning compression using the standard Accept-Encoding and Content-Encoding headers of HTTP.
A similar, but lesser, comment could be made in this specification concerning charset encoding using the standard Accept-Charset header of HTTP.

The results of a query may take various formats, and the format does not depend on the input query method used to call the service. The standard format, which must be supported by all services, is the VOTable-v1.1 format as described below. In addition, services may support any number of other formats (e.g. CSV or TSV, XML …).

The output format desired by the client shall be indicated using the standard HTTP mechanism and MIME types. As such, the client uses the "`Accept:`" HTTP header to indicate a list of preferred MIME types for the output.

For example:

```
Accept: application/x-votable+xml, text/csv, */*
```

It might not always be possible to control the output format in this way (for example, when using the asynchronous query method, or when using a web browser to construct and submit query URLs manually).

In such cases, an OUTPUT parameter may be used to force the output format selection. This parameter may be used with all forms of query method, and if present always overrides the "`Accept:`" HTTP header value.

The service must respond to a valid query, that does not generate any error, with a "`200 OK`" status code and the "`Content-Type:`" HTTP header indicating the MIME type of the actual output. (NB: if this conflicts with the UWS specification in the asynchronous case, the behaviour should conform to the UWS specification).

For example:

```
HTTP/1.1 200 OK
Content-Type: application/x-votable+xml
```

If both the OUTPUT parameter and the "`Accept:`" HTTP header are missing from the client call, the default output format is the VOTable-v1.1 format as described below. The "`Content-Type:`" HTTP header shall still be present.

Here we briefly discuss a new idea for empty output results handling. The advantages of this solution are that the mechanism remains consistent regardless of the requested result format, and that server/client processing power is spared and bandwidth usage is reduced. The disadvantage is that current clients may not check for this status code. An alternative approach is to return an empty VOTable (or equivalent for other output formats), as is the case in the SSAP specification.

If the client query produces no result data, the server should respond with a "`204 No Content`" status code and no message body. This behaviour is preferred to returning an empty VOTable (or equivalent empty file for other output formats). Note that this is **not an error status code**; rather, it indicates that the query was valid, and ran to completion without error, but produced no data.

3.1 VOTable output format

This output format is requested using:

```
OUTPUT=VOTABLE
```

Results returned in this output format must comply with the VOTable-v1.1 specification and validate against the official VOTable-v1.1 XSD schema. The MIME type for this format is: `application/x-votable+xml`

The `VOTABLE` must contain a single `RESOURCE` element identified by the attribute `type="results"`, containing a single `TABLE` element with the resulting data.

This specification does not enforce any type of information to be returned, but for each one (each column), the `FIELD` element must have one "`name`" and one "`datatype`" attribute with valid values. Furthermore, it is strongly recommended to fill out the "`ucd`", "`utype`", "`unit`" and "`arraysize`" attributes with valid values whenever possible. A short `DESCRIPTION` element may also prove to be very useful to clients.

3.2 Other output formats

The following list of output formats is by no mean compulsory for services to be compliant, but if they are needed (or rendered compulsory by a service or protocol definition that expands on this one) they must comply with the following definitions. It is strongly discouraged to implement a similar output format with any other MIME type.

3.2.1 The CSV format

This output format is requested using:

```
OUTPUT=CSV
```

Results returned in this output format must comply with the RFC 4180, which can be found here (ref. [6]). The MIME type for this format is: `text/csv`

This format can be useful as it is very simple and easily supported by clients.

3.2.2 The XML format

This output format is requested using:

```
OUTPUT=XML
```

Results returned in this output format must be compliant XML 1.0 documents. The MIME type for this format is: `text/xml`

This format can be useful as it allows complex data structures.

4 Metadata Access Format

Metadata is accessed directly from the service endpoint and should contain all the information needed to construct simple or complete queries. The metadata query methods for an input query method must be supported if the input query method is supported by the service.

Note: Metadata handling is still a very active topic within the IVOA. The notes in this section may be superseded by the VOSI specification, which covers the standard interfaces (including metadata access methods) that should be supported by a VO-compliant service. The metadata access methods to be supported by this protocol should be either pure VOSI or a superset of VOSI.

4.1 *Metadata query methods*

The Metadata Query methods are invoked via a standard **HTTP/1.1 GET** request to a URL endpoint. The URL endpoint takes the following general form:

```
http://metadata.endpoint.{s,c}aq
                [{?,&}table=table_name]
```

The different elements constituting the URL endpoint are:

- The HTTP protocol designation => `http://`
- The specific metadata endpoint => `metadata.endpoint.{s,c}aq`
- The endpoint/parameters separator => `{?,&}`
- The optional table name designator => `table=table_name`

Note that the specific service endpoint may include login information (e.g. "`user:pass@`"), port number (e.g. "`:8080`") and an arbitrary long path. Anchor pointers (e.g. "`#anchor`") may be present in the URL endpoint, but they have no specific meaning in this protocol specification for the service.

The endpoint/parameters separator may be either a question mark or an ampersand depending on the presence or not of a question mark in the specific metadata endpoint.

The optional "`table_name`" parameter value must be URL encoded as it would be if sent from an HTML form with the "`enctype`" set to "`application/x-www-form-urlencoded`".

The HTTP request is then:

```
GET /path[{?,&}table=table_name] HTTP/1.1
Host: metadata.endpoint.url
```

The other HTTP verbs (POST, PUT, DELETE, OPTIONS, HEAD …) are undefined for this query method.

4.1.1 Simple query metadata

Accessing a Simple Query method metadata is done via two unique endpoints.

```
http://metadata.endpoint.saq
```

This shall return the information on input parameters that can be used for this service, encoded in the XML "params" structure described hereafter.

```
http://metadata.endpoint.saq{?,&}table=results
```

This shall return the information on the data returned by this service, encoded in the XML “table” structure described hereafter.

4.1.2 Complete query metadata

Accessing a Complete Query method metadata is done via two unique endpoints. The second endpoint has a variable part informed by a call to the first one.

```
http://metadata.endpoint.caq
```

This shall return the information on the database structure of this service, encoded in the XML “dataset” structure described hereafter.

```
http://metadata.endpoint.caq{?,&}table=table_name
```

This shall return the information on the particular structure of the table “`table_name`” of this service, encoded in the XML “table” structure described hereafter.

If the table “`table_name`” does not exist for this service, the server shall return an error code “`400 Bad Request`” or “`404 Not Found`”.

4.2 Metadata output formats

All metadata output shall conform to the following formats and validate against its respective XSD schema.

4.2.1 The “params” metadata

TODO (to be worked out in coordination with the Registry WG)

4.2.2 The “dataset” metadata

TODO (to be worked out in coordination with the Registry WG)

4.2.3 The “table” metadata

TODO (to be worked out in coordination with the Registry WG)

5 Error Responses

Hereafter, we briefly discuss a new idea for error handling. The advantage of this solution is that the same format of error response is used, regardless of the requested output format. The disadvantage is that the list of error codes cannot be extended. An alternative approach is to return a VOTable-based error output, as is the case in the SSAP specification.

Error output shall be supported using the standard error codes from the HTTP protocol. Here is a list of correspondences, and explanations of the different codes.

<u>Here are the client error codes:</u>

- 400 Bad Request – used for malformed or erroneous input query
- 401 Unauthorized – reserved for error accessing proprietary data
- 403 Forbidden – reserved for blocked access to proprietary data
- 404 Not Found – used for queries to an unsupported method

<u>Here are the server error codes:</u>

- 500 Internal Server Error – general miscellaneous server error
- 501 Not Implemented – unimplemented optional behaviour/parameter
- 502 Bad Gateway – backend error (database, store, external service)
- 503 Service Unavailable – temporary problem with server or backend
- 504 Gateway Timeout – backend not responding or timing out

It is strongly encouraged to use the message body to return a textual explanation of the error for the client/user. The description should be plain UTF-8 encoded English text, without any formatting tags, destined for a human reader.

Compliance with DAL Services

Following this protocol paradigm, any DAL protocol could be defined as an n-step protocol made up of:

- an initial step using the above described protocol spec for the queryData
- a second step (e.g. SIAP, SSAP, …) making use of the getData capability of the service
- possible further n-steps (to follow eventually from here if required in future)

Further considerations on these types of protocol are deferred to an eventual discussion within the DAL Working Group.

References

[1] VOQL-TEG, *Astronomical Data Query Language (ADQL)*,
http://www.ivoa.net/Documents/latest/ADQL.html

[2] ISO, *Database Language SQL*,
http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt

[3] D. Tody et al., *Simple Image Access Protocol (SIAP)*,
http://www.ivoa.net/Documents/latest/SIA.html

[4] D. Tody et al., *Simple Spectral Access Protocol (SSAP)*,
http://www.aoc.nrao.edu/~dtody/ssa/ssa-v097.pdf

[5] F. Ochsenbein et al., *VOTable Format Definition*,
http://www.ivoa.net/Documents/latest/VOT.html

[6] Y. Shafranovich, *Comma-Separated Values Files (rfc4180)*,
http://www.ietf.org/rfc/rfc4180.txt

[7] G. Rixon, *Universal Worker Service (UWS)*,
http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/UWS-0.3.pdf