



International

Virtual

Observatory

Alliance

Table Access Protocol

Version 1.0

IVOA Proposed Recommendation 2009 October 6

Interest/Working Group:

<http://www.ivoa.net/cgi-bin/twiki/bin/view/IVOA/IvoaDAL>

This version:

PR-TAP-1.0-20091006

Latest version:

<http://www.ivoa.net/Documents/TAP/>

Previous version(s):

<http://www.ivoa.net/Documents/TAP/20090608/>

<http://www.ivoa.net/internal/IVOA/TableAccess/TAP-0.5.pdf>

<http://www.ivoa.net/internal/IVOA/TableAccess/TAP-0.42-20090420.pdf>

<http://www.ivoa.net/internal/IVOA/TableAccess/TAP-0.41-20090317.pdf>

<http://www.ivoa.net/internal/IVOA/TableAccess/TAP-0.4-20090212.pdf>

<http://www.ivoa.net/internal/IVOA/TableAccess/TAP-0.31-20081124.pdf>

<http://www.ivoa.net/internal/IVOA/TableAccess/TAP-v0.3.pdf>

Authors:

P. Dowler, G. Rixon, D. Tody

Editors:

P. Dowler

Contributors:

Table Access Protocol

K. Andrews, J. Good, R. Hanisch, G. Lemson, T. McGlynn, K. Noddle, F. Ochsenbein, I. Ortiz, P. Osuna, R. Plante, J. Salgado, A. Stebe, A. Szalay

Abstract

The table access protocol (TAP) defines a service protocol for accessing general table data, including astronomical catalogs as well as general database tables. Access is provided for both database and table metadata as well as for actual table data. This version of the protocol includes support for multiple query languages, including queries specified using the Astronomical Data Query Language (ADQL) and the Parameterised Query Language (PQL) within an integrated interface. It also includes support for both synchronous and asynchronous queries. Special support is provided for spatially indexed queries using the spatial extensions in ADQL. A multi-position query capability permits queries against an arbitrarily large list of astronomical targets, providing a simple spatial cross-matching capability. More sophisticated distributed cross-matching capabilities are possible by orchestrating a distributed query across multiple TAP services.

Status of This Document

This is an IVOA Proposed Recommendation made available for public review.

It is appropriate to reference this document only as a recommended standard that is under review and which may be changed before it is accepted as a full recommendation.

A list of [current IVOA Recommendations and other technical documents](http://www.ivoa.net/Documents/) can be found at <http://www.ivoa.net/Documents/>.

Contents

1	Introduction.....	5
1.1	Query Types.....	5
1.1.1	Data Queries.....	5
1.1.2	Metadata Queries.....	5
1.1.3	VOSI	5
1.2	Query Languages.....	6
1.2.1	ADQL Queries.....	6
1.2.2	PQL Queries.....	6
1.2.3	Other Query Languages.....	6
1.3	Query Execution.....	6
1.3.1	Asynchronous Queries.....	7
1.3.2	Synchronous Queries.....	7

Table Access Protocol

1.4 Interface Overview (informative).....	7
2 Requirements for a TAP service (normative).....	10
2.1 Feature Overview.....	10
2.2 Web resources.....	11
2.2.1 /sync.....	11
2.2.2 /async.....	11
2.2.3 /availability.....	12
2.2.4 /capabilities.....	12
2.2.5 /tables.....	13
2.3 Parameters for HTTP requests.....	13
2.3.1 REQUEST.....	13
2.3.2 VERSION.....	14
2.3.3 LANG.....	14
2.3.4 QUERY.....	14
2.3.5 Parameters for PQL.....	15
2.3.6 FORMAT.....	15
2.3.7 MAXREC.....	16
2.3.8 RUNID.....	16
2.3.9 UPLOAD.....	17
2.3.10 Case of parameters.....	17
2.3.11 Order and cardinality of parameters.....	17
2.4 Table names.....	17
2.5 Table Upload.....	17
2.5.1 UPLOAD.....	20
2.5.2 Inline Table Upload.....	20
2.6 Metadata Tables and TAP schema.....	21
2.6.1 Schemas.....	21
2.6.2 Tables.....	21
2.6.3 Columns.....	22
2.6.4 Foreign Keys.....	22
2.7 Representations of results.....	23
2.7.1 Data and metadata queries.....	23
2.7.2 VOSI.....	24
2.7.3 Errors.....	24
2.7.4 Overflows.....	25

Table Access Protocol

2.8	Versioning of the TAP protocol.....	25
2.8.1	Version number form and value.....	25
2.8.2	Appearance in requests and in service metadata.....	25
2.8.3	Version number negotiation.....	26
2.9	Use of VOTable.....	26
2.9.1	INFO elements.....	26
2.9.2	Version Mismatch Errors.....	28
3	Service Registration (normative).....	29
4	Extended capabilities (normative).....	30
5	Use of UWS (informative).....	31
5.1	Creating a Query.....	31
5.2	Running the Query.....	32
6	VOSpace Integration (informative).....	34
7	Use of HTTP (informative).....	35
7.1	General HTTP request rules.....	35
7.1.1	Introduction.....	35
7.1.2	Reserved characters in HTTP GET URLs.....	35
7.1.3	HTTP GET.....	36
7.1.4	HTTP POST.....	37
7.2	General HTTP response rules.....	37
8	References.....	38

1 Introduction

The *Table Access Protocol* (TAP) is a Web-service protocol that gives access to collections of tabular data referred to collectively as a *tableset*. TAP services accept queries posed against the *tableset* available via the service and return the query response as another table, in accord with the relational model. Queries may be submitted using various query languages and may execute synchronously or asynchronously. Support for the Astronomical Data Query Language (ADQL) is mandatory; support for other query languages such as Parameterised Query Language (PQL) or native SQL is optional.

The result of a TAP query is another table, returned as a VOTable or in some other format. Support for VOTable output is mandatory; all other formats are optional.

The table collections made accessible via TAP are typically stored in relational database management systems (RDBMS). A TAP service exposes the database schema to client applications so that queries can be posed directly against arbitrary data tables available via the service.

Multi-table operations such as joins or cross matches are possible provided the tables are all managed by the local TAP service, and provided the service supports these capabilities. Larger scale operations such as a distributed cross match are also possible, but require combining the results of multiple TAP services.

1.1 Query Types

TAP services support three kinds of queries: data queries, metadata queries, Virtual Observatory Service Interface (VOSI [6]) queries.

1.1.1 Data Queries

Data queries apply to the astronomical content served by a TAP service. This is the reason for providing a TAP service. All the other kinds of query support the ability to make data queries. Data queries may be specified in any query language supported by the service.

1.1.2 Metadata Queries

Metadata queries work like data queries, using the same query languages, but they are applied to standardized tables (a subset and patterned after *information schema* in RDBMS) which explain the data model of a particular TAP service. Metadata queries allow a client to discover the names of tables and columns to be used in data queries.

1.1.3 VOSI

Virtual Observatory Service Interface (VOSI [6]) specifies base service interface common to all VO services. VOSI requests supply metadata concerning the availability ('VOSI-availability') of a TAP service, its main interfaces ('VOSI-

capabilities'), and its data model ('VOSI-tables'). VOSI-capabilities and VOSI-tables outputs use the same XML schema as the IVOA registry and can be incorporated in service registrations.

1.2 Query Languages

TAP supports the use of multiple query languages, some of which are described here.

1.2.1 ADQL Queries

Support for ADQL queries is mandatory. ADQL can be used to specify queries that access one or more tables provided by the TAP service, including the standard metadata tables. In general, the client must access table metadata in order to discover the names of tables and columns and then formulate queries. ADQL queries provide a direct (low-level) access to the tables; a query will be written for a specific TAP service and will not be usable with other services unless they implement the exact same data model. It is also possible that the service registration (in an IVOA Registry) may include sufficient table metadata to enable queries to be written directly.

Details of the ADQL language may be found in [1].

1.2.2 PQL Queries

Support for PQL is optional. PQL can be used to formulate queries that access a single table provided by the TAP service, including the standard set of metadata tables. Since PQL is more abstract than ADQL, it can also be used in some cases without first querying the metadata tables by using the PQL parameters which carry sufficient meaning to enable the service to decide which tables and columns to use (e.g. POS, SIZE, REGION, BAND, TIME).

Details of the PQL language (parameters) is not part of the TAP specification.

1.2.3 Other Query Languages

A TAP service may also support use of other query languages, including pass-through of native SQL directly to an underlying DBMS, by describing such capabilities in the service metadata and allowing custom values of the service parameters. This mechanism allows future developments within the VOQL Working Group and outside the IVOA to be used without revising the TAP specification.

1.3 Query Execution

The TAP service specification defines both synchronous and asynchronous query execution. Users select synchronous or asynchronous execution by choosing the appropriate resource below the base URL for the service (see 2.2). A query is synchronous if the results of the query are delivered in the HTTP response to the request that originally posed the query. If the service returns an immediate HTTP-response upon accepting a query and the client later obtains

the results of the query in response to a separate HTTP request, then we say the request is asynchronous.

1.3.1 Asynchronous Queries

Asynchronous query support is mandatory. Asynchronous queries require that client and server share knowledge of the state of the query during its execution and between HTTP exchanges. They are an example of stateful interactions. In TAP, the mechanism by which the clients and services share the state of transactions is based on the Universal Worker Service (UWS) pattern [3].

1.3.2 Synchronous Queries

Synchronous query support is mandatory. Synchronous queries execute immediately and the client must wait for the query to finish. If the HTTP request times out or the client otherwise loses the connection to the service before receiving the response, then the query fails.

Synchronous query execution is adequate when the query will execute quickly and with a small number of results, or when they can at least start returning results quickly. They are generally simple to implement using standard web technologies and easy to use from a browser or scripting environment. However, synchronous queries are generally not sufficient and likely to fail for queries that take a long time to execute, especially before returning any results.

1.4 Interface Overview (informative)

Table Access Protocol (TAP) is implemented over the HTTP protocol using standard HTTP GET and POST requests and conventions. A TAP request specifies one or more parameter key/value pairs; both keys and values are strings. The keys used are discussed in this specification and in the specifications for query languages supported by a service. The values may need to be encoded, using standard URL-encoding to replace non-alphanumeric characters with %xx sequences. For the following examples, <http://example.com/tap/> is the base URL for a TAP service.

This is an example¹ of a synchronous ADQL query on *r* magnitude:

```
HTTP POST http://example.com/tap/sync
REQUEST=doQuery
LANG=ADQL
QUERY=SELECT * FROM magnitudes as m where m.r>=10 and m.r<=16
```

The equivalent PQL query would be:

```
HTTP POST http://example.com/tap/sync
REQUEST=doQuery
LANG=PQL
```

1 Throughout this document, example URL parameters are shown without URL-encoding for clarity; ADQL queries in particular must be encoded since they have spaces and various special characters.

Table Access Protocol

```
FROM=magnitudes  
WHERE=r,10/16
```

Synchronous queries return the table of results in the HTTP response² to the initial request. In the examples above, the output format defaults to VOTable; the FORMAT parameter could be added to select a different format.

Asynchronous queries are started in the same way as the synchronous kind, using the /async endpoint:

```
http://example.com/tap/async  
REQUEST=doQuery  
LANG=ADQL  
QUERY=SELECT * FROM magnitudes AS m WHERE m.r>=10 AND m.r<=16
```

or

```
http://example.com/tap/async  
REQUEST=doQuery  
LANG=PQL  
FROM=magnitudes  
WHERE=r,10/16
```

The service's response to these requests is a URL representing the query's state and progress and where the state may be monitored and controlled. The query result or an error document can then be retrieved from a URL associated with the job. This is an application of the UWS pattern.

Positional queries have special support in PQL. This is a cone search on a specified table:

```
http://example.com/tap/sync  
REQUEST=doQuery  
LANG=PQL  
POS=12,34  
SIZE=0.5  
FROM=foo
```

In addition to the sync and async resources for query execution, a TAP service also has metadata resources defined by the VOSI standard. The availability of a service can be monitored by accessing:

```
http://example.com/tap/availability
```

See 2.2.3 for details of the availability resource.

The complete table metadata can be obtained:

```
http://example.com/tap/tables
```

See 2.2.5 for details of the table metadata resource.

The capabilities can be obtained by:

```
http://example.com/tap/capabilities
```

² Synchronous requests may issue a redirect to the result using HTTP code 303 (see other).

Table Access Protocol

The capabilities are also accessible via a service request to the synchronous query resource:

```
http://example.com/tap/sync  
REQUEST=getCapabilities
```

This output lists support for optional TAP functionality and additional implemented interfaces. See 2.2.4 for details.

2 Requirements for a TAP service (normative)

The keywords “must”, “required”, “should”, and “may” as used in this document are to be interpreted as described in the W3C specifications (IETF RFC 2119 [2]). Mandatory interface elements are indicated as **must**, recommended interface elements as **should**, and optional interface elements as **may** or simply “may” without the bold face font.

2.1 Feature Overview

An implementation of a TAP service provides features as follows.

Feature	Resource	Resource support	Parameters	Parameter support
synchronous query execution	/sync	must		
asynchronous query execution	/async	must		
availability (VOSI)	/availability	should		
capabilities (VOSI)	/capabilities	must with / async	REQUEST=getCapabilities	must with / sync
table metadata (VOSI)	/tables	should		
ADQL queries			REQUEST=doQuery LANG=ADQL	must
PQL queries			REQUEST=doQuery LANG=PQL	may
other query languages			REQUEST=doQuery LANG=<other>	may
table upload			UPLOAD	may
VOTable output			FORMAT	must
other formats			FORMAT	should
limiting output			MAXREC	must
logging			RUNID	should

Table Access Protocol

The resources and parameters are described in detail below. The description of these resources and parameters spell out how the requirements here are to be implemented.

TAP service registration in the IVOA resource-registry is specified in section 3 .

2.2 Web resources

A TAP service **must** be represented as a tree structure of web resources each addressable via a URL in the http scheme, or the https scheme, or both.

The web resource at the root of the tree **must** represent the service as a whole. This specification defines no standard representation for this root resource. Implementations **may** provide a representation, or may return a '404 not found' response to requests for the root web-resource. One possible representation is an HTML page describing the scientific usage and content of the service. TAP clients **must not** depend on a specific representation of the root web-resource.

2.2.1 /sync

A TAP service **must** provide a web resource with relative URL */sync* that is a direct child of the root web resource. This web resource represents the results of synchronous requests. The exact form of the query, and hence the representation of the resource, is defined by the query parameters as listed in section 2.3. Representations of results of queries and VOSI outputs are defined in sections 2.7.1 and 2.7.2 respectively.

An HTTP-GET request to the */sync* web resource **may** return a cached copy of the representation. This cached copy might come from an HTTP cache between the client and the service, and the service **may** also maintain its own cache. Clients which require an up-to-date representation of volatile data or metadata must use HTTP POST.

2.2.2 /async

A TAP service **must** provide a web resource with relative URL */async* that is a direct child of the root web resource. This web resource represents controls for asynchronous queries. Specifically, the web resource **must** represent the job-list as specified in the UWS standard [5].

The child web resources of the */async* resource are as specified by UWS. These are descendants of the */async* web-resource, and they include a web resource that represents the eventual result of an asynchronous query. A client making an asynchronous request must use the UWS facilities to monitor or control the job, e.g.:

`http://example.com/tap/async/42/results/result`

where the base URL for the TAP service is:

`http://example.com/tap`

the UWS job list is:

`http://example.com/tap/async`

Table Access Protocol

and the job resource is

`http://example.com/tap/async/42`

where 42 is the job identifier. In addition to the job list and job resource above, UWS specifies the name and semantics of the a small set of child resources used to view and control the job, e.g.:

`http://example.com/tap/async/42/phase`

`http://example.com/tap/async/42/quote`

`http://example.com/tap/async/42/termination`

`http://example.com/tap/async/42/destruction`

`http://example.com/tap/async/42/error`

`http://example.com/tap/async/42/params`

`http://example.com/tap/async/42/results`

Successful TAP queries produce results which **must** be accessible as resources under the UWS result list, e.g.:

`http://example.com/tap/async/42/results/`

Failed TAP queries produce an error document (see 2.9) which **must** be accessible as the error resource, e.g.:

`http://example.com/tap/async/42/error`

The number of results (resources) in the list depends on the query language used. For example, a successful ADQL query will produce a single result file; if the query returned no rows, the result file should exist and contain no data rows. Details on interacting with these resources is specified in the UWS standard; for examples specific to TAP see Section 5 below.

2.2.3 /availability

The VOSI availability metadata **should** be accessible from a web resource with relative URL `/availability` that is a direct child of the root web resource. This resource only supports the http GET method. The content is described by [6].

Services which do not implement the `/availability` resource **must** respond with an HTTP response code of 404 when this resource is accessed.

2.2.4 /capabilities

The service capabilities **should** be accessible from a web resource with relative URL `/capabilities` that is a direct child of the root web resource. Services which implement the `/async` resource **must** also implement the `/capabilities` resource. This resource only supports the http GET method. The content is described by [8].

Services which do not implement the `/capabilities` resource **must** respond with an HTTP response code of 404 when this resource is accessed.

2.2.5 /tables

The table metadata **should** be accessible from a web resource with relative URL /tables that is a direct child of the root web resource. This resource only supports the http GET method. The content is described by [7].

Services which do not implement the /tables resource **must** respond with an HTTP response code of 404 when this resource is accessed.

2.3 Parameters for HTTP requests

The /sync and /async web-resources **must** accept the parameters listed in the following sub-sections. In a synchronous request, the parameters select the representation returned in the response message. In an asynchronous request, the parameters select the representation of the eventual query-result rather than the response to the initial request.

Not all combinations of the parameters are meaningful. For example, if a request carries LANG=ADQL then the *SELECT* parameter (from PQL) is spurious. If a service receives a spurious parameter in an otherwise correct request, then the service **must** ignore the spurious parameter, must respond to the request normally and **must not** report errors concerning the spurious parameter.

2.3.1 REQUEST

This parameter distinguishes current service operations, makes it possible to extend the service spec (with additional or custom operations), and specifies how other parameters should be interpreted. A TAP client **must** set this parameter correctly in every request (GET or POST) to the /async or /sync web resources. If a TAP service receives a request without this parameter or with an incorrect value for this parameter, then the service **must** reject the request and return an error document as the result.

These are the standard values of the parameter:

- doQuery: create or execute a query
- getCapabilities: return VOSI-capabilities metadata

All requests to create (/async) or execute (/sync) a query using a query language **must** include REQUEST=doQuery and **must** include the LANG parameter. For other values of REQUEST, additional parameters may or may not be required. The REQUEST=getCapabilities service operation **must** be supported for synchronous (/sync) requests and is not defined for asynchronous (/async) requests.

For synchronous queries, the HTTP request **must** also include additional parameters (see below) with the details of the query. This is used for metadata queries and data queries.

For asynchronous queries, the additional parameters may be included with the HTTP request that creates the query (the UWS job) or they may be POSTed

directly to the created job resource, in one or more separate HTTP requests. The parameter names remain the same in both cases.

2.3.2 VERSION

The *VERSION* parameter specifies the TAP protocol version number. The format of the version number, and version negotiation, are described in section 2.9.2 .

A TAP service **must** support the *VERSION* parameter.

2.3.3 LANG

The LANG parameter specifies the query language. The service **must** support *LANG* and the client **must** provide a value with REQUEST=doQuery. The only standard values for the *LANG* parameter is ADQL (a required language) and PQL (reserved value for an optional language which is under development). Support for other languages and the *LANG* value to use with them is described in the service capabilities.

For example, an ADQL query would be performed with

```
REQUEST=doQuery
LANG=ADQL
QUERY=<ADQL query string>
```

A PQL query would be performed with

```
REQUEST=doQuery
LANG=PQL
<PQL-specific parameters>
```

The value of LANG is a string specifying the language and optionally the language version used for the subsequent query parameter(s), as defined by the service capabilities. The client **may** specify the version of the query language, e.g. LANG=ADQL-2.0 (the syntax should be as shown) or it may omit the version, e.g. LANG=ADQL. The service **should** return an “unknown query language” error as described in 2.9 if an unsupported language or an incompatible language version is specified.

2.3.4 QUERY

The QUERY parameter is used to specify the ADQL query. It may also be used to specify the query for other values of LANG (e.g. LANG=<some RDBMS-specific SQL variant>) which are not specified in this document but may be described in the service capabilities.

A service **must** support the *QUERY* parameter because ADQL is a required language. The case sensitivity of the query string is defined solely by the query language specification. In the case of ADQL 2.0, for example, the query is not case sensitive except for character literals; schema, table, and column names, function names, and other ADQL keywords are not case sensitive.

Within the ADQL query, the service **must** support the use of timestamp values in ISO8601 format, specifically yyyy-MM-dd[‘T’HH:mm:ss[.SSS]], where square

brackets denote optional parts and the 'T' denotes a single character separator (T) between the date and time parts.

If the tables that are queried through a service contain columns with spatial coordinates and the services supports spatial querying via the ADQL “region” constructs, the service **must** support the *INTERSECTS* function and it **must** support the following geometry functions: *REGION*, *POINT*, *BOX*, *CIRCLE*, *COORD1*, *COORD2*, *COORDSYS*. Support for the *AREA*, *CONTAINS*, and *POLYGON* functions are optional. If the service supports the *REGION* function, it **must** support region encoding in STC-S format [4]; the extent of STC-S support within the *REGION* function is left up to the implementation. Coordinate system specification for *POINT*, *BOX*, *CIRCLE*, and *POLYGON* **must** use values from Table 3 (standard reference frames) in STC [4].

Note: Although it is allowed by the ADQL syntax, clients should be careful when mixing constants and column references for coordinate system and coordinate values. For example, POINT('ICRS', t.ra, t.dec) does not cause t.ra and t.dec to be transformed to ICRS; it simply tells the service to treat the values as being expressed in that coordinate system.

2.3.5 Parameters for PQL

A number of parameters are defined by PQL for use in parametric queries. All of the parameters for PQL are specified in elsewhere and are used unchanged in TAP.

Within the PQL query, the service **must** support the use of timestamp values in ISO8601 format (see 2.3.4).

If the table that is queried contains columns with spatial coordinates and the services wants to enable the caller to perform spatial queries, the service **must** support the PQL spatial constraint parameters (POS,SIZE and REGION). If a service supports the REGION parameter, it **must** support region encoding in STC-S format [4]; the extent of STC-S support within the *REGION* function is left up to the implementation. Coordinate system qualifiers **must** use values from Table 3 (standard reference frames) in STC [4].

PQL defines symbolic values (@something). In TAP these can be used to refer to an uploaded table (see 2.5) with parameters that support a table reference. When used in this way, the uploaded table must be treated as if in the TAP_UPLOAD schema(e.g. @TAP_UPLOAD.mytable). As with all query languages, details on how to use table references in PQL are not specified here.

2.3.6 FORMAT

The *FORMAT* parameter indicates the client's desired format for the table of results of a query. Its value **should** be a MIME type for tabular data or one of the following shorthand forms:

table type	mime type(s)	short form

Table Access Protocol

VOTable	application/x-votable+xml text/xml	votable
comma separated values	text/csv	csv
tab separated values	text/tab-separated-values	tsv
FITS binary table	application/fits	fits
pretty-printed text	text/plain	text
pretty-printed Web page	text/html	html

All the shorthand forms are insensitive to case. If the *FORMAT* parameter is omitted, the default format is VOTable.

A TAP service **must** support VOTable as an output format, **should** support CSV and TSV output and **may** support other formats. A TAP service **must** accept a *FORMAT* parameter indicating a format that the service supports and **should** reject queries where the *FORMAT* parameter specifies a format not supported by the service implementation.

2.3.7 MAXREC

The service **must** accept a *MAXREC* parameter specifying the maximum number of table records (rows) to be returned. If *MAXREC* is not specified in a query, the service **may** apply a default value or **may** set no limit. If the result set for a query exceeds this value, the service **must** only return the requested number of rows. If the result set is truncated in this fashion, it must include an overflow indicator as specified in section 2.7.4 .

The service **must** support the special value of *MAXREC=0*. *This value* indicates that, in the event of an otherwise valid request, a valid output table be returned containing metadata, no table data rows, and an overflow indicator as specified in section 2.7.4 . A query with *MAXREC=0* can be used with a simple query (e.g. `SELECT * FROM some_table`) to extract and examine the VOTable metadata (assuming *FORMAT=votable*). Note: in this version of TAP, this is the only mechanism to learn some of the detailed metadata, such as coordinate systems used.

2.3.8 RUNID

The service **should** implement the *RUNID* parameter, used to tag service requests with the job ID of a larger job of which the request may be part. The *RUNID* parameter is defined in [3] for */async* requests; services should also implement it for */sync* requests.

For example, if a cross match portal issues multiple requests to remote TAP services to carry out a cross-match operation, all would receive the same

RUNID, and the service logs could later be analyzed to reconstruct the service operations initiated in response to the job.

The service **should** ensure that *RUNID* is preserved in any service logs and **should** pass on the *RUNID* value in any calls to other services.

2.3.9 UPLOAD

The service **should** support table upload via the *UPLOAD* parameter. The value is a list of table-name,URI pairs. Table names must be legal ADQL table names as defined in [1]. URIs maybe be simple URLs (e.g. scheme is http) or URIs (e.g. scheme is vos or param) that give the location of the the table content. See section 2.5 for details.

2.3.10 Case of parameters

Parameter names **must not** be case sensitive, but parameter values **must** be case sensitive. In this document, parameter names are typically shown in uppercase for typographical clarity, not as a requirement.

2.3.11 Order and cardinality of parameters

Parameters in a request **may** be specified in any order.

When request parameters are duplicated with conflicting values, the response from the service is undefined. The service **may** reject the request or it **may** pick one value for the parameter. Clients **should not** repeat parameters in a request.

2.4 Table names

A fully qualified table name has the form

```
[[catalog_name".".]schema_name".".]table_name
```

where *catalog_name* is the the name of the DB catalogue (often the “database” name) in SQL DBMS terminology, *schema_name* is the name of the “schema” in DBMS terminology (often also called a “database”; a DBMS schema is a type of data model where the top level data model elements are tables), and *table_name* is the actual table name. All elements of the table name are optional except *table_name*. Depending upon the DBMS, “catalog” or “schema” may or may not be implemented; some DBMS implement both, others one or the other, and the simplest database systems might not implement either.

The implementation of a TAP service **must** define the table names acceptable in queries and **must** reveal these to clients through metadata queries or through VOSI-tables output, and the names **must** be identical in each of these sources. A TAP client must determine the acceptable names from one of these sources or from the cached form of the VOSI-tables output included in the service's registration.

2.5 Table Upload

The service **should** implement the table upload capability. If upload is supported, the service must accept tables in VOTable format. The client specifies the name

Table Access Protocol

of uploaded table; this name **must** be legal ADQL table name with no catalog or schema (e.g. an unqualified table name). Uploaded tables **must** be referred to in queries as *TAP_UPLOAD.<tablename>*.

Tables in the *TAP_UPLOAD* schema are transient and persist only for the lifetime of the query (although caching might be used behind the scenes) and are never visible in the *TAP_SCHEMA* metadata.

The column names in the transient database table are taken directly from the name attribute of the VOTable FIELD and PARAM elements. The datatypes of the transient table are determined from the FIELD and PARAM attributes as follows:

VOTable: datatype	VOTable: arraysize	VOTable: xtype	ADQL: column type
boolean	[1]		Not supported
short	[1]		SMALLINT
int	[1]		INTEGER
long	[1]		BIGINT
float	[1]		REAL
double	[1]		DOUBLE
<numeric type>	> 1		VARBINARY
char	[1]		CHAR(1)
char			VARCHAR ³
char	n*		VARCHAR(n)
char	n		CHAR(n)
unsignedByte			VARBINARY ⁴
unsignedByte	n*		VARBINARY(n)
unsignedByte	n		BINARY(n)
unsignedByte	n, *, n*	adql:BLOB	BLOB
char	n, *, n*	adql:CLOB	CLOB
char	n, *, n*	adql:TIMESTAMP	TIMESTAMP
char	n, *, n*	adql:POINT	POINT
char	n, *, n*	adql:REGION	REGION

Table Access Protocol

The default mapping of data types are shown above (no arraysize or xtype). If the xtype attribute is set, this is the preferred internal datatype. If xtype is not set, then the datatype and arraysize indicate the most suitable internal datatype.

In the arraysize column above, [1] means the arraysize is not set or is set to 1, n means arraysize is set to a specific value, * means arraysize="*", and n* means arraysize="n*" (variable size up to length n). A blank means the arraysize is not set.

Binary values (unsignedByte in VOTable, BINARY, VARBINARY, or BLOB in ADQL) can be expressed as specified by the VOTable standard. By default, VOTable allows them to be written as an array of decimal numbers, e.g. 12 56 0 255 0 0 255 (one number per byte value).

TIMESTAMP values are specified using ISO8601 format without a timezone (as in 2.3.4) and are assumed to be in UTC. The xtype="adql:TIMESTAMP" attribute must be specified in an uploaded VOTable in order for the values to be inserted in a column of type TIMESTAMP; without the xtype, the values would be inserted into a CHAR(n) or VARCHAR column.

POINT and REGION values are specified in STC-S format (as in 2.3.4). The xtype="adql:POINT" attribute must be specified in an uploaded VOTable in order for the values to be treated as POINTs (e.g. to be used with some of the ADQL region functions). For regions, the xtype="adql:REGION" attribute must be specified in an uploaded VOTable in order for the values to be treated as REGIONs (e.g. to be used with some of the ADQL region functions).

2.5.1 UPLOAD

The *UPLOAD* parameter is used to reference read-only external tables via their URI, to be uploaded for use as input tables to the query. The value of the *UPLOAD* parameter is a list of table name-URI pairs. Elements of the list are delimited by semicolon and the two parts of the pair are delimited by comma. For example:

```
UPLOAD=table_a,http://host_a/path;table_b,http://host_b/path
```

would define two input tables *table_a* and *table_b*, located at the given URIs. Services that implement *UPLOAD* **must** support *http* as a URI scheme (e.g. must support treating an *http* URI as a URL). A VOSpace URI (*vos:<something>*) is a more generic example of a URI that requires more service-side functionality; support for the *vos* scheme is optional.

3 This is the default internal datatype for character values. The service implementation must chose a suitable size for the VARCHAR column

4 This is the default internal datatype for binary values. The service implementation must chose a suitable size for the VARBINARY column

2.5.2 Inline Table Upload

To upload a table inline caller must specify the UPLOAD parameter (as above) using a special URI scheme “param”. This scheme indicates that the value after the colon will be the name of the inline content. The content type used is *multipart/form-data*, using a “file” type input element. The “name” attribute must match that used in the UPLOAD parameter.

For example, in the POST data we might have this parameter:

```
UPLOAD=table_c,param:table1
```

and this content:

```
Content-Type: multipart/form-data; boundary=AaB03
[...]
--AaB03x
Content-disposition: form-data; name="table1"; filename="table1.xml"
Content-type: application/x-votable+xml
[...]
--AaB03x
[...]
```

The uploaded table would be referenced in queries as *TAP_UPLOAD.table_c* (the table name in the UPLOAD parameter). Services that implement table upload **must** support the *param* scheme for inline uploads.

In principle, any number of tables can be uploaded using the UPLOAD parameter and any combination of URI schemes supported by the service as long as they are assigned unique table names within the query. Services may limit the size and number of uploaded tables; if the service refuses to accept the entire table it **must** respond with an error as described in 2.7.3 .

2.6 Metadata Tables and TAP schema

The TAP core schema defines a set of tables in the TAP_SCHEMA schema that contain the minimal metadata required to describe and use the tables exposed by a TAP service. Services must provide these tables and make them accessible by all supported query mechanisms. The information in the TAP core-schema is equivalent to that defined by VOSI-tables and allowed by the registry for a *VODataService* [7].

The qualified names in the tables of the TAP schema **must** follow the rules defined in section 2.4. The names **must** be stated in a form that is acceptable as an operand of a query. The TAP_SCHEMA may be queried for tables named TAP_SCHEMA.* to get information about the schema itself, e.g., to determine if any extended schema metadata is defined by the service.

All columns in the TAP_SCHEMA tables are of type VARCHAR except for size, principal, indexed, and std (in Columns) which are INTEGER values.

2.6.1 Schemas

The table TAP_SCHEMA.schemas **must** contain the following columns:

Table Access Protocol

Column name	datatype	
schema_name	varchar	fully qualified schema name
description	varchar	brief description of schema
utype	varchar	UTYPE if schema corresponds to a data model

The schema_name values are unique. The fully qualified schema name is defined by the ADQL language and follows the pattern *[catalog.]schema*.

2.6.2 Tables

The table TAP_SCHEMA.tables **must** contain the following columns:

Column name	datatype	
schema_name	varchar	fully qualified schema name
table_name	varchar	fully qualified table name
table_type	varchar	one of: table, view
description	varchar	brief description of table
utype	varchar	UTYPE if table corresponds to a data model

The table_name values are unique. The fully qualified table name is defined by the ADQL language and follows the pattern *[[catalog.]schema.]table*. Services should specify such table names so that clients can use them directly in queries.

2.6.3 Columns

The table TAP_SCHEMA.columns must contain the following columns:

Column name	datatype	
table_name	varchar	fully qualified table name
column_name	varchar	column name
description	varchar	brief description of column
unit	varchar	unit in VO standard format
ucd	varchar	UCD of column if any
utype	varchar	UTYPE of column if any
datatype	varchar	ADQL datatype as in section 2.5
size	integer	length of variable length datatypes ⁵
principal	integer	a principal column; 1 means true, 0 means false
indexed	integer	an indexed column; 1 means true, 0 means false
std	integer	a standard column; 1 means true, 0 means

⁵ Variable length datatypes include CHAR, VARCHAR, BINARY, and VARBINARY. The size should be NULL for fixed size datatypes (numeric, timestamp) and may be NULL for arbitrary-sized columns (CLOB, BLOB).

Table Access Protocol

		false
--	--	-------

The table_name,column_name (pair) values are unique.

2.6.4 Foreign Keys

The table TAP_SCHEMA.keys **must** contain the following columns to describe foreign key relations between tables:

Column name	datatype	
key_id	varchar	unique key identifier
from_table	varchar	fully qualified table name
target_table	varchar	fully qualified table name
description	varchar	description of this key
utype	varchar	utype of this key

The key_id values are unique and used only to join with the TAP_SCHEMA.key_columns table below. There may be one or more rows with different key_id values and a pair of tables to denote one or more ways to join the tables.

The table TAP_SCHEMA.key_columns **must** contain the following columns to describe the columns that make up a foreign key :

Column name	datatype	
key_id	varchar	key identifier from the TAP_SCHEMA.keys
from_column	varchar	key column name in the <from_table>
target_column	varchar	key column in the <target_table>

There may be one or more rows with a specific key_id to denote single or multi-column keys.

A TAP service **must** provide the tables listed above and **may** provide other tables in the TAP_SCHEMA namespace.

Data types and how they map to VOTable datatypes are described in section 2.5 above. The “size” gives the length of variable length datatypes, for example varchar(256); this size does not map to the VOTable arraysize attribute when the latter specifies the size and shape of a multi-dimensional array. The “principal” flag indicates that the column is considered a core part the content; clients can use this hint to make the principal column(s) visible, for example by selecting them by default in generating an ADQL query. In cases where the services chose the columns to return (such as PQL without a SELECT parameter), the principal column indicates which columns are returned by default. The “indexed” flag indicates that the column is indexed, potentially making queries run much faster if this column is used in a constraint. The “std” is included for compatibility with the registry, which uses this value to indicate that a given column is defined by some standard, as opposed to a custom column defined by a particular service.

2.7 Representations of results

2.7.1 Data and metadata queries

The result of a data query or a metadata query depends on the query language used and may be one or more tables in one or more files. Unsupportable combinations of query result and FORMAT (e.g. queries that produce multiple tables and an inherently single-table format like CSV) will cause the request to fail. Currently, an ADQL query result **must** be a single table (in a single file).

This table **must** be encoded in the output format specified by the FORMAT parameter of the query. See section 2.3.6 for required, optional and default formats. VOTable is the default format and VOTable support is mandatory.

The output table **must** include the same number and order of columns as specified in the SELECT clause of the query. For VOTable output, the name attribute of FIELD elements **must** be the same as the column names (or aliases if specified in the query) from the query and the datatype, arraysize, and xtype attributes of FIELD elements **must** be set using the mapping specified in section 2.5. The xtype attribute in the output **must** match the datatype for the column in the TAP_SCHEMA.

VOTable structure follows the rules in section 2.9 and **must** be returned with an allowed VOTable MIME type (*application/x-votable+xml* or *text/xml*). If the FORMAT parameter (see 2.3.6) of the request specified a specific VOTable mime type, the requested mime type **must** be used in the HTTP response.

CSV formatted data **should** represent the output table with one row of text per table row, with the table column values rendered as text and separated by commas. If a column value contains a comma the entire column value **should** be enclosed in double quotes. Text lines may be arbitrarily long. The first data row **should** give the column name as the data value. CSV data **must** be returned with a MIME type of *text/csv*; if the optional header line (with column names) is included, the MIME type must be *text/csv;header=present*. Full details of CSV format are defined in RFC 4180 [14].

TSV formatted data **should** represent the output table with one row of text per table row, with the table column values rendered as text and separated by the TAB character. TSV data **must** be returned with a MIME type of *text/tab-separated-values* [15]. Column values may not contain the TAB character.

2.7.2 VOSI

Representations of VOSI outputs (capabilities, availability, table metadata) **must** be as defined in the VOSI standard [6].

The representation of table metadata **must** include all tables in the service's tableset. VOSI's representation of table metadata is specified in *VODataService* [7].

2.7.3 Errors

If the service detects an exceptional condition, it **must** return an error document with an appropriate HTTP-status code. TAP distinguishes three classes of exceptions.

- Errors in the use of the HTTP protocol.
- Errors in the use of the TAP protocol, including both invalid requests and failure of the service to complete valid requests.

Error documents for HTTP-level errors are not specified in the TAP protocol. Responses to these errors are typically generated by service containers and cannot be controlled by TAP implementations. There are several cases where a TAP service could return an HTTP error. First, the /async endpoint could return a 404 (not found) error if the client accesses a job within the UWS joblist that does not exist. Second, access to a resource could result in an HTTP 401 (not authorized) error if authentication is required or an HTTP 403 (forbidden) error if the client is not allowed to access the resource.

Error documents for TAP errors **must** be VOTable documents; any result-format specified in the request is ignored. If the error document is being retrieved from the /async/<jobid>/error resource (specified by UWS) after an asynchronous query, the HTTP status code should be 200. If the error document is being returned directly after a synchronous query, the service may use an appropriate HTTP status code, including 200 (successfully returning a response to the request) and various 4xx and 5xx values. The exception condition **must** be described to the client using a status code in the VOTable header. Section 2.9 specifies the use of VOTable for error documents in TAP services.

2.7.4 Overflows

If a query is executed by a TAP service, the number of rows in the table of results may exceed a limit requested by the user (using the MAXREC parameter) or a limit set by the service implementation (the default or maximum value of MAXREC). In these cases, the query is said to have 'overflowed'. Typically, a TAP service will not detect an overflow until some part of the table of results has been sent to the client.

If an overflow occurs, the TAP service **must** produce a table of results that is valid, in the required output format, and which contains all the results up to the point of overflow. Since an output overflow is not an error condition, the MIME type of the output **must** be the same as for any successful query and the HTTP status-code **must** be as for a successful, complete query.

If the output format is VOTable, section 2.9.1 describes the method by which the overflow is reported. No method of reporting an overflow is defined for formats other than VOTable.

2.8 Versioning of the TAP protocol

The TAP protocol provides explicitly for versioning of the interface in order to support version negotiation between a client and a service where one or both parties support more than one version.

2.8.1 Version number form and value

The TAP protocol defines a protocol version-number. The version number applies to all aspects of the protocol as defined in this document, including any associated XML schema and the request encodings. The TAP version refers only to the TAP protocol; query languages are versioned separately and TAP and ADQL versions may differ.

Version numbers follow IVOA document conventions.

2.8.2 Appearance in requests and in service metadata

The version number may appear in at least three places: in the service metadata, as a parameter in client requests to a server, and in the query response. The version number used in a client's request of a particular server must be equal to a version number which that server has declared it supports (except during negotiation, as described below). A server may support several versions, whose values clients may discover according to the negotiation rules.

2.8.3 Version number negotiation

If a TAP client does not specify the version number in a request, the server assumes the highest standard version supported by the service, and no explicit version checking takes place. If the client specifies an explicit version number, and this does not match a version available from the service at level two, the service returns a version number mismatch error as described in 2.9.2. The client can determine what versions of the protocol the service supports by a prior call to VOSI-capabilities or via a registry query.

2.9 Use of VOTable

VOTable is a general format. TAP requires that it be used in a particular way.

The result VOTable document **must** comply with VOTable v1.2 or greater [9]. For columns containing coordinate values, the coordinate system metadata should be provided as described in [13].

The VOTable **must** contain a *RESOURCE* element identified with the attribute *type="results"*, containing a single *TABLE* element with the results of the query. Additional *RESOURCE* elements may be present, but the usage of any such elements is not defined here and TAP clients **should not** depend upon them.

2.9.1 INFO elements

The *RESOURCE* element **must** contain, before the *TABLE* element, an *INFO* element with attribute *name = "QUERY_STATUS"*. The *value* attribute **must** contain one of the following values:

Table Access Protocol

- “OK”, meaning that the query executed successfully and a result table is included in the resource
- “ERROR”, meaning that an error was detected at the level of the TAP protocol or the query failed to execute

The DESCRIPTION element conveying the status **should** be a message suitable for display to the user describing the status.

Examples:

```
<INFO name="QUERY_STATUS" value="OK"/>
<INFO name="QUERY_STATUS" value="OK">
  <DESCRIPTION>Successful query</DESCRIPTION>
</INFO>
<INFO name="QUERY_STATUS" value="ERROR">
  <DESCRIPTION>value out of range in POS=45,91</DESCRIPTION>
</INFO>
```

Additional *INFO* elements **may** be provided, e.g., to echo the input parameters back to the client in the query response (a useful feature for debugging or to self-document the query response), but clients **should not** depend on these.

Example:

```
<RESOURCE type="results">
<INFO name="QUERY_STATUS" value="ERROR">
  <DESCRIPTION>unrecognized operation</DESCRIPTION>
</INFO>
<INFO name="SPECIFICATION" value="TAP"/>
<INFO name="VERSION" value="1.0"/>
<INFO name="REQUEST" value="doQuery"/>
<INFO name="baseUrl" value="http://webtest.aoc.nrao.edu/ivoa-dal"/>
<INFO name="serviceVersion" value="1.0"/
...
</RESOURCE>
```

If an overflow occurs (result exceeds MAXREC), the service must close the table and append another INFO element to the RESOURCE (after the TABLE) with *name="QUERY_STATUS"* and the *value="OVERFLOW"*.

Table Access Protocol

Example:

```
<RESOURCE type="results">
<INFO name="QUERY_STATUS" value="OK"/>
...
<TABLE>...</TABLE>
<INFO name="QUERY_STATUS" value="OVERFLOW"/>
</RESOURCE>
```

In the above example, the TABLE should have exactly MAXREC rows.

If an error occurs while writing the rows of the VOTable, the service must close the table and append another INFO element to the RESOURCE, after the TABLE, with *name="QUERY_STATUS"* and the *value="ERROR"*.

Example:

```
<RESOURCE type="results">
<INFO name="QUERY_STATUS" value="OK"/>
...
<TABLE>...</TABLE>
<INFO name="QUERY_STATUS" value="ERROR" />
</RESOURCE>
```

The content of these trailing INFO elements is optional and intended for users; client software **should not** depend on it.

Thus, one INFO element with *name="QUERY_STATUS"* and *value="OK"* or *value="ERROR"* **must** be included before the TABLE. If the TABLE does not contain the entire query result, one INFO element with *value="OVERFLOW"* or *value="ERROR"* **must** be included after the table.

2.9.2 Version Mismatch Errors

Errors due to version mismatch from either the VERSION parameter (TAP version) or specific version used in the LANG parameter (query language version) are specified using an INFO element with *name="QUERY_STATUS"* and *value="ERROR"* as described above.

3 Service Registration (normative)

Publication of a service to the VO requires that it be registered with an IVOA registry, including describing the identity and capabilities of the service.

The resource document for a TAP service instance **must** be structured according to *VOResource* 1.0 [8] using the sub-type *CatalogService* as defined in *VODataService* 1.1 [7].

The resource document **must** include a *capability* element denoting the TAP interface and functions. This element must contain the URL for the root web resource (as defined in section 2.2). Clients would add to this URL */sync* or */async* as appropriate.

The resource document **must** contain capability elements for the VOSI-capabilities, VOSI-availability and VOSI-tables outputs. These **must** be formatted as in the VOSI standard [6].

The resource document should include the table metadata, except where the database-schema of the archive changes frequently.⁶ Where table metadata are provided, they **must** be represented as XML elements drawn from *VODataService* 1.1.

6 If the database schema changes faster than the changes can be propagated through the publishing registries to the full registries, then it is pointless to register the table metadata. If the details change hourly then clearly the registries cannot keep up; if the details change yearly, then clearly they can. Intermediate cases are less certain, but weekly changes are probably too fast and monthly changes probably slow enough.

4 Extended capabilities (normative)

The TAP service allows for optional extended capabilities and operations. Extensions may be defined within an information community when needed for additional functionality or specialization. A generic client **must** not be required or expected to make use of such extensions. Extended capabilities or operations **must** be defined by the service metadata. Extended capabilities provide additional metadata about the service, and may or may not enable optional new parameters to be included in operation requests. Extended operations may allow additional operations to be defined.

A server **must** produce a valid response to the operations defined in this document, even if parameters used by extended capabilities are missing or malformed (i.e. the server **must** supply a default value for any extended capabilities it defines), or if parameters are supplied that are not known to the server.

Service providers **must** choose extension names with care to avoid conflicting with standard metadata fields, parameters and operations.

5 Use of UWS (informative)

The UWS pattern is specified in [3] and its application to TAP in section 2.2.2. This section explains the exchange of messages between a TAP client and service when using UWS to run an asynchronous query.

Consider a TAP service at *http://example.com/tap*. TAP mandates that the asynchronous requests be directed to *http://example.com/tap/async*. This URL points to the list of 'jobs'; i.e. the list of queries currently or recently executed.

5.1 Creating a Query

To create a new query, the client POSTs a request to the job list:

```
HTTP POST http://example.com/tap/async
REQUEST=doQuery
LANG=ADQL
QUERY=SELECT TOP 100 * FROM foo
```

The service then creates a job and assigns that job a name and a URL based on the name. Suppose that the name is 42, then the URL will be *http://example.com/tap/async/42* because the jobs are always children of the job list. While the job is in the PENDING phase, additional parameters may be specified by additional POSTs to the job resource, for example:

```
HTTP POST http://example.com/tap/async/42
UPLOAD=mytable,http://a.b.c/mytable.xml
```

After each such POST, the service issues an HTTP redirection to the job's URL, where the modified state may be accessed:

```
HTTP status 303 'See other'
Location: http://example.com/tap/async/42
```

All TAP-specific parameters are stored using the paramList mechanism of UWS and are included in the XML representation of the job:

```
HTTP GET http://example.com/tap/async/42
```

or directly from the params resource:

```
HTTP GET http://example.com/tap /async/42/params
```

Individual parameters cannot be accessed as separate web resources.

The UWS pattern requires the following resources to describe and control the job:

```
http://example.com/tap/async/42/phase
http://example.com/tap/async/42/quote
http://example.com/tap/async/42/termination
http://example.com/tap/async/42/destruction
http://example.com/tap/async/42/results
http://example.com/tap/async/42/error
```

The quote resource specifies the predicted completion time for the job (query), assuming it is started immediately. Practically, it is very hard to estimate the time

Table Access Protocol

a query will take; for TAP services it is recommended that this be set to the current time plus the maximum amount of time the query will be allowed to run (see termination below).

The termination resource specifies the amount of time (in seconds) the job (query) will be allowed to run before being aborted by the service. The termination time is set by the service and can be read from the job or directly from the termination resource:

```
HTTP GET http://example.com/tap/async/42/termination
```

The service may allow the client to change the termination:

```
HTTP POST http://example.com/tap/async/42/termination
```

```
TERMINATION=600
```

The destruction resource specifies when the service will destroy the job. The service is only required to keep a job for a finite period of time, after which it may destroy the job, including the result. After this time, the client will receive an HTTP 404 'not found' status if it tries to get any information about the job. The destruction time of the job is chosen by the service and the client can read it from the job or directly from the destruction resource:

```
HTTP GET http://example.com/tap/async/42/destruction
```

The service may allow the client to change the destruction time:

```
HTTP POST http://example.com/tap/async/42/destruction
```

```
DESTRUCTION=2008-11-11T11:11:11Z
```

5.2 Running the Query

The *phase* URL shows the progress of the job. When the job is created by the service it will normally be set to *PENDING*, but might be set to *ERROR* if the service has rejected the job. If the phase is *ERROR*, then the *error* URL should lead to an error document explaining the problem. If the phase is *PENDING*, then the client needs to commit the job for execution.

The client **runs** the job by posting to the phase URL:

```
HTTP POST http://example.com/tap /async/42/phase
```

```
PHASE=RUN
```

The service replies with a redirection to the job URL

```
HTTP status 303 'see other'
```

```
Location: http://example.com/tap /async/42
```

The phase will now have changed to either *QUEUED* or *EXECUTING*, depending on the service implementation. The client **tracks** the execution by polling the phase URL:

```
HTTP GET http://example.com/tap/async/42/phase
```

A job in the *QUEUED* or *EXECUTING* phase may be **aborted** by posting to the phase URL:

```
HTTP POST http://example.com/tap/async/42/phase
```

```
PHASE=ABORT
```

Table Access Protocol

When the query is complete, the phase changes to *COMPLETED*. The client then retrieves the result from the results list:

```
HTTP GET http://example.com/tap/async/42/results/result
```

The client knows that the table of results is at the URL `/result` relative to the results list because the TAP protocol requires this naming. A generic UWS client could find the name of the result and retrieve it by examining either the job description:

```
HTTP GET http://example.com/tap/async/42
```

or by looking specifically at the result list:

```
HTTP GET http://example.com/tap/async/42/results
```

If the service cannot run the query, then the final phase is *ERROR* and there is no table of results. In this case, the client should expect an HTTP 404 'not found' status if it tries to retrieve the result. The client should look instead at the error resource to find out what went wrong:

```
HTTP GET http://example.com/tap/async/42/error
```

If the job was aborted (by the client or the service), the final phase will be *ABORTED* and there is no table or results. As with errors, the client should look at the error resource to find out what went wrong.

The basic sequence can be executed from a web browser or from a shell script using the *curl* utility:

```
curl -d 'REQUEST=doQuery&LANG=PQL&POS=12,34&SIZE=0.5&FROM=foo' \  
      http://example.com/tap/async  
[read Location header from curl output]  
curl -d 'PHASE=RUN' http://example.com/tap/async/42  
curl http://example.com/tap/async/42/phase  
[repeat until phase is COMPLETED]  
curl http://example.com/tap/42/results/result
```

6 VOspace Integration (informative)

This version of TAP provides limited VOspace integration, although better support for VOspace is planned for a later version following prototyping. In this version, one may specify an upload table using a URI to a table stored in a VOspace, e.g.:

```
HTTP POST http://example.com/tap/async/42
UPLOAD=mytable,vos://space/path/votable.xml
```

The service would resolve the URI, contact the VOspace, retrieve the table, and make it visible to the query as TAP_UPLOAD.mytable.

A future version of TAP may specify additional use and more integration with VOspace.

7 Use of HTTP (informative)

A TAP service is a web service and TAP implementations are constrained by the general rules for use of HTTP, which are contained in IETF RFC documents. This section collates some of the requirements. For authoritative specifications, please refer to the original RFCs.

7.1 General HTTP request rules

7.1.1 Introduction

This document defines the implementation of the TAP service on a distributed computing platform (DCP) comprising Internet hosts that support the Hypertext Transfer Protocol (HTTP) (see IETF RFC 2616 [11]). Thus, the Online Resource of each operation supported by a server is an HTTP Uniform Resource Locator (URL). The URL may be different for each operation, or the same, at the discretion of the service provider. Each URL **must** conform to the description in IETF RFC 2616 (section 3.2.2 “HTTP URL”) but is otherwise implementation-dependent; only the query portion comprising the service request itself is defined by this document.

While the TAP protocol currently only supports HTTP as the DCP for general Parameterised operations, data access references are more general and may use other internet protocols, e.g., FTP, or potentially grid protocols.

HTTP supports two primary request methods: GET and POST. One or both of these methods may be offered by a server, and the use of the Online Resource URL differs in each case. Support for the GET method is mandatory; support for the POST method is optional except where required for a service operation to function, e.g., uploading a large quantity of data inline in a query, or when issuing a request to the service which changes the server state.

7.1.2 Reserved characters in HTTP GET URLs

The URL specification (IETF RFC 2396 [5]) reserves particular characters as significant and requires that these be escaped when they might conflict with their defined usage. This document explicitly reserves several of those characters for use in the query portion of TAP requests. When the characters “?”, “&”, “=”, “,” (comma), “/”, and “;” appear in one of the roles defined in Table 1, they **must** appear literally in the URL. When those characters appear elsewhere (for example, in the value of a parameter), they should be encoded as defined in IETF RFC 2396. The server **must** be prepared to decode any character escaped in this manner.

Table 1 — Reserved characters in TAP query string

Character	Reserved usage
?	Separator indicating start of query string.
&	Separator between parameters in query string.

Table Access Protocol

=	Separator between name and value of parameter.
,;	Separator between individual values in list-oriented parameters

In particular, if any parameter value contains the character “#” (for example in a dataset identifier) it must be URL encoded to be legally included in a URL.

7.1.3 HTTP GET

A TAP service **must** support the “GET” method of the HTTP protocol (IETF RFC 2616 [11]).

An Online Resource URL intended for HTTP GET requests is in fact only a URL prefix to which additional parameters are appended in order to construct a valid Operation request. A URL prefix is defined in accordance with IETF RFC 2396 [5] as a string including, in order, the scheme (“http” or “https”), Internet Protocol hostname or numeric address, optional port number, path, mandatory question mark “?”, and optional string comprising one or more server-specific parameters ending in an ampersand “&”. The prefix defines the network address to which request messages are to be sent for a particular operation on a particular server. Each operation may have a different prefix. Each prefix is entirely at the discretion of the service provider.

This document defines how to construct a query part that is appended to the URL prefix in order to form a complete request message. Every TAP operation has several mandatory or optional request parameters. Each parameter has a defined name . Each parameter may have one or more legal values, which are either defined by this document or are selected by the client based on service metadata. To formulate the query part of the URL, a client **must** append the mandatory request parameters, and any desired optional parameters, as name/value pairs in the form “name=value&” (parameter name, equals sign, parameter value, ampersand). The “&” is a separator between name/value pairs, and is therefore optional after the last pair in the request string.

When the HTTP GET method is used, the client-constructed query part is appended to the URL prefix defined by the server, and the resulting complete URL is invoked as defined by HTTP (IETF RFC 2616).

Table 2 summarizes the components of an operation request URL when HTTP GET is used.

Table 2 — Structure of TAP request using HTTP GET

URL component	Description
http://host:port/path[?[name=[value]]&name=[value]]	Base-URL (prefix) of service operation. [] denotes 0 or 1 occurrence of an optional part; {} denotes 0 or more occurrences.
name=value&	One or more standard request parameter name/value pairs as defined for each operation by this document.

7.1.4 HTTP POST

TAP uses the “POST” method of the HTTP protocol (IETF RFC 2616 [11]) for submitting query parameters to both the /async and /sync endpoints and for modifying the state of a job on the /async endpoint. In addition, POST must be used to upload a table inline as described in 2.5.2 . Parameters should be URL encoded in a POST whenever they would need to be URL encoded for a GET.

7.2 General HTTP response rules

Upon receiving a valid request, the server **must** send a response corresponding exactly to the request as detailed in section 2.7 of this document, or send a service exception if unable to respond correctly. Only in the case of Version Negotiation (see 2.8.3) may the server offer a differing result. Upon receiving an invalid request, the server **must** return an error document as described in section 2.7.3 .

A server may send an HTTP Redirect message (using HTTP response codes as defined in IETF RFC 2616 [11]) to an absolute URL that is different from the valid request URL that was sent by the client. HTTP Redirect causes the client to issue a new HTTP request for the new URL. Several redirects could in theory occur. Practically speaking, the redirect sequence ends when the server responds with a valid TAP response. The final response **must** be a TAP response that corresponds exactly to the original request (or a service exception).

Response objects **must** be accompanied by the appropriate Multipurpose Internet Mail Extensions (MIME) type (IETF RFC 2045 [12]) for that object. A list of MIME types in common use on the internet is maintained by the Internet Assigned Numbers Authority (IANA) . Allowable types for operation responses and service exceptions are discussed in 2.7.1 .

Response objects should be accompanied by other HTTP entity headers as appropriate and to the extent possible. In particular, the Expires and Last-Modified headers provide important information for caching; *Content-Length* may be used by clients to know when data transmission is complete and to efficiently allocate space for results, and *Content-Encoding* or *Content-Transfer-Encoding* may be necessary for proper interpretation of the results.

8 References

TODO: update 3, 6, 7, and 9 when relevant RFC+TCG+exec process is complete.

- [1] I. Ortiz, J. Lusted, P. Dowler, A. Szalay, Y. Shirasaki, M. Nieto- Santisteban, M. Ohishi, W. O'Mullane, P. Osuna, VOQL-TEG & VOQL-WG, *IVOA Astronomical Data Query Language version 2*, IVOA recommendation 30th October 2008.
<http://www.ivoa.net/Documents/REC/ADQL/ADQL-20081030.pdf>
- [2] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, IETF RFC 2119.
<http://www.ietf.org/rfc/rfc2119.txt>
- [3] P. Harrison & G. Rixon, *Universal Worker Service Version 1.0*, IVOA Proposed Recommendation, 09 September 2009.
<http://www.ivoa.net/Documents/UWS/20090909/PR-UWS-1.0-20090909.html>
- [4] A. Rots, *Space-Time Coordinate Metadata for the Virtual Observatory Version 1.33*, IVOA Recommendation 30 October 2007.
<http://www.ivoa.net/Documents/REC/DM/STC-20071030.html>
- [5] T. Berner-Lee, R. Fielding L. Masinter, *Uniform Resource Identifiers (URI): Generic Syntax*, IETF RFC 2396.
<http://www.ietf.org/rfc/rfc2396.txt>
- [6] GWS-WG, G. Rixon (ed.), *IVOA Support Interfaces Version 1.00*, IVOA Working Draft, 25 August 2009.
<http://www.ivoa.net/Documents/VOSI/20090825/WD-VOSI-1.0-20090825.html>
- [7] R. Plante (ed.), A. Stébé, K. Benson, P. Dowler, M. Graham, G. Greene, P. Harrison, G. Lemson, A. Linde, G. Rixon & IVOA Registry-WG, *VODataService: a VOResource Schema Extension for Describing Collections and Services Version 1.1*. IVOA Proposed Recommendation, 03 September 2009
<http://www.ivoa.net/Documents/VODataService/20090903/PR-VODataService-1.1-20090903.html>
- [8] R. Plante (ed.), K. Benson, M. Graham, G. Greene, P. Harrison, G. Lemson, A. Linde, G. Rixon, A. Stébé, & IVOA Registry-WG, *VOResource: an XML Encoding Schema for Resource Metadata Version 1.03*, IVOA Recommendation 22 February 2008.
<http://www.ivoa.net/Documents/REC/ReR/VOResource-20080222.html>
- [9] F. Ochsenbein (ed.), R. Williams, *VOTable Format Definition Version 1.2*, IVOA Proposed Recommendation 29 September 2009.
<http://www.ivoa.net/Documents/VOTable/20090929/PR-VOTable-1.2-20090929.html>
- [10] P. Biron & A. Malhotra, *XML Schema Part 2: Datatypes Second Edition*, W3C Recommendation 28 October 2004.
<http://www.w3.org/TR/xmlschema-2/>
- [11] R. Fielding, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, IETF RFC 2616.
<http://www.rfc-editor.org/rfc/rfc2616.txt>

Table Access Protocol

- [12] N. Freed & N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*, IETF RFC 2045.
<http://www.ietf.org/rfc/rfc2045.txt>
- [13] F. Ochsenein, J. McDowell, A. Rots, *Referencing STC in VOTable Version 1.1*, IVOA Note 12 June 2009.
<http://www.ivoa.net/Documents/Notes/VOTableSTC/NOTE-VOTableSTC-1.1-20090612.html>
- [14] Y. Shafranovich, *Common Format and MIME Type for Comma-Separated Values (CSV) Files*, IETF RFC 4180. <http://www.ietf.org/rfc/rfc4180.txt>
- [15] IANA, *MIME Media Types*,
<http://www.iana.org/assignments/media-types/text/tab-separated-values>