International

Virtual

Observatory

Alliance

Table Access Protocol

Version 0.4

IVOA Internal Working Draft 2009 February 12

This version:

TAP-V0.4-20090212

Latest version:

Not yet issued

Previous version(s):

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

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

http://www.ivoa.net/internal/IVOA/TableAccess/tap-v0.2.pdf

http://www.ivoa.net/internal/IVOA/TableAccess/TAP-QL-0.1.pdf

Lead authors:

P. Dowler, G. Rixon (editor), D. Tody

Contributors:

K. Andrews, J. Good, R. Hanisch, T. McGlynn, K. Noddle,

F. Ochsenbein, I. Ortiz, P. Osuna, R. Plante, G. Rixon, 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 a working draft internal to the DAL-WG.

This is an IVOA Working Draft for review by IVOA members and other interested parties. It is a draft document and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use IVOA Working Drafts as reference materials or to cite them as other than "work in progress".

A list of current IVOA Recommendations and other technical documents can be found at http://www.ivoa.net/Documents/.

Acknowledgements

"Ack here, if any"

Contents

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), but TAP may also be implemented for data stored in other ways, such as in flat-file systems. This aspect of the implementation is abstracted by the protocol and is not visible to users. 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 Types of query

1.1.1 Data, metadata, tableset and VOSI queries

TAP services distinguish four different kinds of query by the information returned.

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.

Metadata queries work like data queries, using the same query languages, but they are applied to standardized tables which explain the data model of a particular TAP installation. Metadata queries allow a client to discover the names of tables and columns to be used in data queries.

Tableset queries are a special case metadata queries that reveal the entire data model in one response. They use a different output format to metadata queries.

VOSI 'queries' supply metadata concerning the 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.

[NOTE: What is the difference between a tableset query and a VOSI-tables query? By implication they both have VOResource output format. -Ed.]

1.1.2 ADQL Queries

Support for ADQL is mandatory. ADQL can be used to query a standard set of metadata tables in order to discover the names of tables and columns, their meanings, data types, and units, and how different tables are related and may be joined. Using this information, queries that target the astronomical content may be correctly written and executed. It is also possible that the service registration (in an IVOA Registry) may include sufficient table metadata to enable queries to be written directly.

1.1.3 PQL Queries

Support for PQL is optional. PQL can be used to query both the standard set of metadata tables and the data tables (as above for 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).

For example, one could perform a simple spatial query without checking the table metadata with:

```
http://some.where/some/thing?POS=180,42&SIZE=0.5
```

This query could be translated as “from the table dataset represented by the service at the URL `http://some.where/some/thing`, find all records for which the recorded position is within 0.5 degrees of the search position (180,42), where the coordinates are right ascension and declination in the ICRS coordinate system, measured in degrees”. Parametric queries such as this are simple to express and to implement for cases where the data model is sufficiently well defined and adequate for the data to be queried, hiding many of the details required to pose and evaluate the query.

1.1.4 Other Query Languages

A TAP service may also support use of other query languages, including pass-through of native SQL directly to the underlying RDBMS, 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.

1.2 Query Execution

The TAP service specification defines synchronous and asynchronous query execution. A query is synchronous if the results of the query are delivered in the HTTP response to the request that originally posed the query. Conversely, 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.

1.2.1 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 (REF).

1.2.2 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.

Metadata queries are always executed synchronously.

1.3 Interface Overview (informative)

TAP defines a RESTful web service with two primary URLs: one for synchronous queries and one for asynchronous queries.[1] The query language and query constraints for a particular request are determined by HTTP request parameters.

This is an example of the URL for a synchronous ADQL query on *r* magnitude:

```
http://some.where/tap/sync?REQUEST=doQuery&LANG=ADQL \
&QUERY='SELECT * FROM magnitudes as m where m.r>10 and
m.r<16'
```

The URL for an equivalent PQL query would be

```
http://some.where/tap/sync?REQUEST=doQuery&LANG=PQL \
&SELECT=$ALL&FROM=magnitudes&WHERE=r,10/16
```

The REQUEST parameter indicates the kind of query required.

1 The original plan was to have just one, primary URL for all kinds of query. After investigation, we find that a single URL cannot satisfy fully the requirements of both the DAL generation-2 conventions and the UWS standard.

Synchronous queries return the table of results in the HTTP response 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.

Synchronous queries can be requested using HTTP GET or HTTP POST; the examples show the GET form. Using GET allows the query result to come from a cache, either inside the TAP service or between the service and client.

Asynchronous queries are started in the same way as the synchronous kind, using the other URL:

```
HTTP POST to http://some.where/tap/async
REQUEST=doQuery
LANG=ADQL
QUERY='SELECT * FROM magnitudes as m where m.r>10 and m.r<16'
```

or

```
HTTP POST to http://some.where/tap/async
REQUEST=doQuery
LANG=PQL
FROM=magnitudes
WHERE=r,10/16
```

The service's response to these requests is an HTTP redirection (code 303 'see other') to a URL representing the query's state and progress. The progress may be tracked by polling a URL for the state: it will eventually go to *COMPLETED* or *ERROR*. The results, or an error document can then by 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://some.where/TAP/sync? \
REQUEST=doQuery&LANG=PQL&POS=12,34&SIZE=0.5&FROM=foo
```

Each TAP service has its own 'tableset': a particular collection of tables and columns with locally-defined names. Those local names are the operands in the queries and so a client needs to know the tableset for a particular service to form a query. There are two ways of exploring the tableset.

First, a description of the entire tableset may be obtained in XML via the VOSI-tables URL, e.g.

```
http://some.where/TAP/sync?REQUEST=getTableMetadata
```

These metadata are in the format defined for the IVOA resource-registry and the client may find a cached copy in the registry.

Secondly, the structure of the tableset is described by a set of tables with fixed names beginning with *TAP_SCHEMA*. These can be queried using either ADQL or the parametric-query languge.

This lists the columns of table 'foo':

```
http://some.where/tap/sync\
?REQUEST=doQuery&LANG=PQL&FROM=TAP_SCHEMA.COLUMNS&WHERE=tablename,foo
```

The service's availability can be read using VOSI:

```
http://some.where/tap/sync?REQUEST=getAvailability
```

Finally, the service's capabilities can be read using VOSI:

```
http://some.where/tap/sync?REQUEST=getCapabilities
```

This output lists any extra interfaces such as legacy cone-search or built-in VOSpace support.

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.

	synchronous	asynchronous	parameters
ADQL	**must**	**must**	REQUEST, LANG
PQL	**may**	**may**	REQUEST, LANG
Other query languages	**may**	**may**	REQUEST, LANG
Service availability (VOSI)	**should**	n/a	REQUEST
Service metadata (VOSI)	**should**	n/a	REQUEST
Table metadata (VOSI)	**should**	n/a	REQUEST

The parameters listed in the last column are described below; the description of these parameters spell out how the requirements here are to be implemented.

A TAP service **must** be registered in the IVOA resource-registry in the form specified in section 3 .

> *[Arguably, the requirements above come into force* because *a service is registered as TAP. This opens the question as to inheritance of requirements when a service derived from TAP is registered in a different form. E.g., a service searching a catalogue image cubes could be defined as a TAP service with a specific data-model and a different form of registration. In this case it is not clear that ADQL query would still be mandatory. This point was discussed briefly at the October 2008 Interop but no conclusion was reached. -Ed.]*

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-source. This web resource represents the results of synchronous queries, including metadata outputs defined by VOSI. 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 data queries, metadata queries and VOSI outputs are defined in sections 2.7.1, 2.7.2 and 2.7.3 respectively.

> *[TBD: The VOSI metadata operations are accessible via the /sync web-resource only. Since both /async and /sync are required for TAP this does not impose any limitation. Future DAL services that follow this pattern would be required to have a /sync web resource even if they do not require or even define synchronous service activity. The alternative would be to access the VOSI metadata directly at the base URL, but that would be incompatible with the DAL2 style embodied in the SSA specification.]*

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].

A TAP service **must** provide web resources 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 and to retrieve the result. If an asynchronous query succeeds, then the table of results **must** be made available as a UWS result with the formal name *result*.[2]

In addition to the web resources specified by UWS, a TAP job also includes the following resources:

```
/async/<jobid>/lang: the query language in use
/async/<jobid>/query: the complete set of query parameters/text
```

2 A 'UWS result' is a web resource for which the position in the tree of web resources is defined by the UWS standard. See section 5 for an explanation in the context of TAP.

```
/async/<jobid>/format: the requested output FORMAT
/async/<jobid>/maxrec: the value of MAXREC that limits the query
/async/<jobid>/mtime: the value of MTIME that limits the query
/async/<jobid>/upload/<table name>
```

Requests to modify these resources and run the query on the */async* web-resource are always sent as HTTP-POST requests.

> *[TBD: It is not obvious what an HTTP GET of an uploaded table, e.g. /async/<jobid>/upload/<table name> should return...]*

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. E.g., 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 query requests from VOSI requests and specifies how other parameters should be interpreted. A TAP client **must** set this parameter correctly in every request. 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 allowed values of the parameter.

- doQuery: create or execute a query
- getCapabilities: return VOSI-capabilities metadata
- getAvailability: return VOSI-availability metadata
- getTableMetadata: return VOSI-tables 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 (VOSI operations) no other parameters are needed and if supplied they **must** be silently ignored.

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

directly to the TAP-specific job resources as described above. The parameter names remain the same in both cases.

> *[The above change to allowed REQUEST values moves the ADQL vs PQL vs other QL into the required LANG parameter below. It preserves the DAL-2 service operation style but decouples it from the query payload.]*

2.3.2 VERSION

The *VERSION* parameter specifies the protocol version number. The format of the version number, and version negotiation, are described in section 2.8.

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. For example, an ADQL query would be performed with

```
REQUEST=doQuery&LANG=ADQL&<ADQL-specific parameters>
```

A PQL query would be performed with

```
REQUEST=doQuery&LANG=PQL-1.0&<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 if an unsupported and incompatible value of *LANG* is specified.

```
Example: put a real LANG error here
```

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 metadata.

A service **must** support the *QUERY* parameter. The query string is case sensitive. In particular, the case of table and column names **must** be preserved between a metadata query and a subsequent query of a data table.

Within the ADQL query, the service **must** support the use of datetime/timestamp values in ISO8601 format.

If the tables that are queried through a service contain columns with spatial coordinates and the services wants to enable the caller to perform spatial queries, 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].

Although it is allowed by the ADQL syntax, services **should** return an error if use of *POINT, BOX, CIRCLE*, or *POLYGON* mix constants and column references for coordinate system and coordinate values. For example, *POINT('ICRS', t.ra, t.dec)* **should** be an error. The only place where mixing constants and column references is acceptable is when using a constant for the radius in a *CIRCLE*, e.g. *CIRCLE(t.coordsys, t.ra, t.dec, 0.1)*, as this allows the caller to control search radii for multi-position searches or source cross-matching.

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 [reference to PQL-0.1-20090212] and are used unchanged in TAP. Specific PQL parameters that are not applicable to the content being queried must be silently ignored. For example, if a PQL query includes TIME=2009-01-01T12:00:00/ and the content includes no time information at all, this parameter must be ignored.

> *[Should this ignoring of parameters be specified here or in PQL? Following the ADQL separation, PQL would specify the params and TAP would specify whether or not to ignore if n/a.]*

Within the PQL query, the service **must** support the use of datetime/timestamp values in ISO8601 format.

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 specify the name of an uploaded table (see 2.3.7) from which a list of values should be taken and used. This mechanism can be used to query with a large list of values than can sensibly be embedded in a query string (using the PQL list syntax). For example, a multi-position search could be performed as

```
Example: REQUEST=doQuery&LANG=PQL \
&POS=@mytable \
&SIZE=0.05 \
&UPLOAD=mytable,<uri to a VOTable>
```

The ability to perform such a multi-valued search is only possible if the service supports table upload.

[TBD: With the above example, the service would have to automagically use the right columns from the uploaded table for values – it may have other columns, especially if it is the output of some other query or service. In table upload, the client specifies the name of the table the service gets column names from the VOTable FIELD elements (also TBD). In ADQL queries, the client has to know the column names as specified in the FIELD elements and use them specifically. Maybe here we should also require the client to be explicit, e.g.

```
&POS=@mytable.ra,@mytable.dec
```

insteading trying to specify how a service should interpret the uploaded table.]

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:

- votable
- csv (comma separated values)
- fits (FITS binary table)
- text (pretty-printed text)
- html (pretty-printed Web page)

All the shorthand forms are insensitive to case.

If the parameter is omitted, the default format is VOTable.

A TAP service **must** support VOTable as an output format, **should** support CSV 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 demands an unsupported format.

2.3.7 UPLOAD

The service **should** implement an *UPLOAD* parameter, used to reference read-only external tables via their URL, to be uploaded for use as input tables to the query. Tables uploaded in this fashion are assumed to be encoded in VOTable format. The value of the *UPLOAD* parameter is a list of table name-URL tuples, delimited by semicolon, using comma to delimit each table name-URL tuple. 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 URLs (URL-encoding is mandatory in this case since we embedding a URL within a URL). The specified table names are arbitrary but **must** be legal ADQL table names and **must** be unique within the upload table namespace for the lifetime of

the query (see section 2.6). The given name for the table name **should** be an unqualified table-name; uploaded tables will automatically be qualified with the schema name T*AP_SCHEMA*. The upload table storage area is shared with any tables uploaded in-line with the query.

> *[Since this is specifically an HTTP REST service, why is the UPLOAD list done with a semi-colon-separated list rather than the normal multi-valued parameter use in HTTP?*
>
> *e.g. UPLOAD=table_a,http://host_a/path&UPLOAD=table_b,http://host_b/path*
>
> *Web technologies and tools handle this transparently.]*

Tables may also be uploaded directly (inline) as described in 2.6 below.

> *[TBD: We should try to consolidate the table upload stuff.]*

2.3.8 MAXREC

The service **should** implement a *MAXREC* parameter specifying the maximum number of table records (rows) to be returned. If the result set for a query exceeds this value a valid data table **should** be returned with an overflow indicator as specified in section 2.7.5 .

If *MAXREC* is not specified in a query, the service **may** apply a default value or **may** set no limit. The default *MAXREC* value defined by a service **should** be large enough to avoid overflow for most small queries, but small enough to provide a response to the user reasonably quickly. The client **may** override the default *MAXREC*, increasing the value up to the maximum value permitted by the service, as defined in the service capabilities. A sufficiently large *MAXREC* may permit streaming of arbitrarily large output tables. Output tables larger than the maximum permitted value of *MAXREC* **must** use some other technique such as asynchronous computation of the output table followed by retrieval using a streaming synchronous GET (VOSpace output may also be supported in a later version of TAP).

A value of *MAXREC=0* indicates that, in the event of an otherwise successful query, a valid output table **should** be returned containing metadata but no table data rows. It is up to the service whether or not to actually execute the query and generate table rows which will be discarded; the query status **should** be returned as “OK” so long as the query is otherwise valid. This is an example of a null query, that is, a query which produces an empty table.

2.3.9 MTIME

The service **may** support an *MTIME* parameter, used to query a table for only rows which were modified within a given range of times, specified as an ISO8601 open or closed range list in the UTC time system. A “modified” row is a table row which was inserted, updated, or deleted during the indicated time interval (hence *MTIME* **may** be used to see deleted rows which are not visible in any other fashion). This feature may be used by a remote client to maintain a replica of a

large table, or to periodically poll a table for changes. The period of time for which deletions are preserved is server dependent (depending upon how often deleted rows are purged) but should be at least one week.

When the *MTIME* parameter is specified, the service **must** add extra columns to the output table (in addition to that specified in the select statement of the query), with utypes *Record.Modified* and *Record.Deleted*. The values in this column are the time-stamp when the last insert/update occurred or when the row was deleted respectively. One value must be set and the other null for every row in the table; values are in ISO8601 format. Rows that are thus marked as deleted must include values for one or more column(s) that uniquely identify the row, but other column values may be null.

The *MTIME* parameter **must not** be used with queries that select from multiple tables. If *MTIME* is used in a such a query the service **must** reject the request and return an error document.

2.3.10 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. 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.

The service **should** pass on the *RUNID* value in any calls to other services, e.g. VOSpace.

2.3.11 Missing or null-valued parameters

If a parameter is not included in a query its value is unset; no value has been specified. If a parameter is given a null value, e.g., “MAXREC=”, the parameter value has been set and the value is the null string. The interpretation of such an input is defined separately for each parameter, and may or may not be an error condition.

> *[TBD: What is the use/value in differentiating between unset and null?]*

2.3.12 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.

2.3.13 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 for the parameter. Clients **should not** repeat parameters in a request.

> *[TBD: Why impose this restriction on multi-valued parameters when it is well defined in HTTP and web technologies handle it seamlessly?]*

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. A client **must** use the names in the exact form given by the service, reserving the case of letters and the embedded punctuation.

2.5 Metadata tables and TAP schema

The TAP core schema is intended to define the minimal metadata required to describe and use the tables exposed by a TAP service. The information in the TAP core-schema is equivalent to that defined by VOSI-tables and allowed by the registry for a *VODataService*.

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

schema_name	fully qualified schema name (catalog.schema)
description	brief description of schema
utype	UTYPE if schema corresponds to a data model

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

schema_name	fully qualified schema name (catalog.schema)

table_name	fully qualified table name (catalog.schema.table)
table_type	one of: base_table, view, output
description	brief description of table
utype	UTYPE if table corresponds to a data model

The table TAP_SCHEMA.columns must contain the following columns:

column_name	column name
table_name	fully qualified table name (catalog.schema.table)
description	brief description of column
unit	unit in VO standard format
ucd	UCD of column if any
utype	UTYPE of column if any
datatype	datatype as in VOTable/Registry
arraysize	array dimensions as in VOTable/Registry
primary	column is visible in default selection
indexed	column is indexed on the server
std	standard column (as opposed to custom)

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

The schema name *TAP_UPLOAD* **should** be included in the table name for any tables uploaded to the service by a client.

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.

The schema naming conventions used here follow that of the registry. Data types are expressed as in VOTable and the registry, e.g., *boolean*, *unsignedByte*, *short*, *int*, *float*, *double*, and so forth. “a*rraysize*” specifies the dimensions of an array, e.g., "*", "5", "5x20" etc. “Primary” indicates that the column should be visible in the default (narrow) view of a table. “Indexed” indicates that the column is indexed, potentially making queries run much faster if this column is used as a constraint. “*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.

The TAP schema also defines *TAP_SCHEMA.tableset*, however this is not an actual table but rather a structured view of the core schema tables above. Special output formats are defined for queries against this view; see section

2.7.2. A simple tableset-query **must** return the entire tableset, but advanced services **may** permit selection with a *WHERE* clause, e.g., to find only tables within a given region or for which the table name matches some pattern.

2.6 Table Uploads

TAP currently supports two methods by which a client application can upload table or other data for use in a query. The simplest approach for tables which are Web-accessible is use of the *UPLOAD* parameter (section 2.3.7) to reference an external table by URI. More flexible for dynamic client queries is the inline table upload where the table is uploaded inline as part of the query.

In both cases uploaded tables share the *TAP_UPLOAD* schema, and **should** be referred to in queries as *TAP_UPLOAD.tablename*, where the *tablename* is specified by the client at upload time, and **must** be a legal ADQL table name. Tables are uploaded in VOTable format. Tables in the *TAP_UPLOAD* schema persist only for the lifetime of the query (although caching might be used behind the scenes).

Uploading a table at query time using the *UPLOAD* parameter is straightforward so long as the table has already been made Web-accessible. For example, a table could be placed in a publicly-readable VOSpace, and the VOSpace URI of the table could be used with *UPLOAD* to reference the table in a query.

In the case of the inline table upload a table is uploaded inline as part of the query, used within the query like any other table, then discarded once the query completes. A typical example would be a multi-position query where the user uploads a list of source positions.

To upload a table inline the POST form of the query **must** be used. The content type used is *multipart/form-data*, using a “file” type input element, with the “name” attribute specifying the table name.

So for example in the POST data (following the header and input parameters) we might have:

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

The uploaded table would automatically propagate and could be referenced in either ADQL or parametric queries as *TAP_UPLOAD.table1*. In the above example a STC region mask is also being uploaded.

Inline table uploads **may** be used both with standard web-forms in a browser, as well as for programmatic input.

Any number of tables can be uploaded using this technique, so long as they are assigned unique table names within the query. Although our discussion here concerns uploading tables, any type of file can be uploaded in this fashion provided the service can do something useful with the file.

2.7 Representations of results

2.7.1 Data and metadata queries

The result of a data query or a metadata query **must** be a single table.

This table **must** be encoded in the output format specified by the FORMAT parameter of the query. See section Error: Reference source not found for required, optional and default formats. VOTable is the default format and VOTable support is mandatory.

VOTables **must** follow the rules in section Error: Reference source not found. These VOTables **should** be returned with a MIME type of *text/xml;content=x-votable*.

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. Header lines **may** optionally be included in the first few lines of output, prior to the first data row, and **should** be indicated by placing the character '#' in the first character of the line.

2.7.2 Tableset queries

If the target of the query is the special table *TAP_SCHEMA.tableset*, then the service **must** support an XML serialization of the tableset and **must** support a special use of VOTable to express the structure of the tableset.

The special, XML serialization **must** conform to the registry standard expressed in *VODataService* v1.1 [7] and the corresponding XML-schema. This serialization format is identical to that used for VOSI tables [6].[3] This format is selected by the parameter setting *FORMAT=xml* in the query.

The special use of VOTable **must** be a data-less VOTable in which the header elements denote the structure of the tableset. There **must** be one *VOTABLE* element per table in the tableset. This is an exception to the rule that query

3 The registry-compliant-XML serialization of the tableset structure is almost the same thing as the VOSI-tables output of the service but is not strictly identical. The format is the same, but while the VOSI output is required to cover all the tables in the tableset (implicitly '*SELECT * FROM TAP_SCHEMA.tableset*), the result of a tableset *query* can be restricted by the *WHERE* clause of that query.

results contain single tables. This format is selected by the parameter setting *FORMAT=votable* in the query.

[In v0.3 of the TAP standard the intent w.r.t metadata queries is clear but the implementation details are not. I have inferred some of the detailed rules as best I can, but may have diverged from the original intent. These details should be cleared up in TAP 0.4 - Ed]

2.7.3 VOSI

Representations of VOSI outputs (service 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 that mandated for the registry in *VODataService* [7].

[In TAP v0.3, it was written that 'The content of the TAP service availability description are TBD.' My understanding of VOSI is that there are no details left to determine. -Ed]

2.7.4 Error documents

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 failure of the service to complete valid requests.
- Overflow conditions where the number of rows returned from a query would exceed a pre-set limit (set either by the client or by the service).

Error documents for HTTP-level errors are not specified in the TAP protocol. Responses to these errors are typically generated by service containers an cannot be controlled by TAP implementations.

Error documents for TAP errors **must** be VOTable documents; in exceptional conditions, any result-format specified in the query is ignored. When returning such a document, the service **must** set HTTP status-code 200 'OK' (because the HTTP operation is correct, even though the request cannot be fulfilled). The exception condition **must** be signaled to the client using a status code in the VOTable header and a qualifier in the MIME type reported in the HTTP header. Section Error: Reference source not found specifies the exact use of these error documents. For Example:

```
<INFO name="QUERY_STATUS" value="ERROR">
DEC out of range: DEC=91
</INFO>
```

Overflow conditions are not strictly errors and results from overflowed queries are not strictly error-documents. Therefore, a response to an overflowed query **must** contain the results table truncated at the row limit and **must** be in the format requested by the client. A response to an overflowed query **should** contain an indication of the overflow if the output format allows this. Section Error: Reference source not found specifies the means of reporting overflows in VOTables. No reporting mechanism is specified for other formats.

2.7.5 Overflows

If a query is executed by a TAP service, the number of rows in the table of results may exceed a limit set by the user (using the MAXREC parameter or the TOP keyword in ADQL) or a limit set by the service implementation. 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.

On detecting an overflow, a TAP service **must** produce a table of results that 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 VOTable **must** be the same as for any successful query and the HTTP status-code **must** be as for a successful, complete query.

If the service detects the overflow before sending the query response to the client, and if the output format is VOTable, then the service **must** include in the table of results an *INFO* element with name attribute set to *QUERY_STATUS* and value element set to *OVERFLOW*. The service should set the value of this element to an error message explaining the overflow. The error message should state the number of rows at which the output was truncated.

If the output format is VOTable, and if the service detects the overflow after the header for the table of results has been sent to the client, then the service **must**, after closing the *TABLE* element for the table of results, write another *TABLE* element to indicate the overflow. This latter table **must** not include data but **must** an *INFO* element announcing the overflow. This element **must** have attributes and content as specified for the case where the overflow was detected before starting to write the VOTable.

```
<INFO name="QUERY_STATUS" value="OVERFLOW">
Number of table rows exceeds default limit of 5000
</INFO>
```

No method of reporting an overflow is defined for formats other than VOTable.

> *[Discussion with FO in Baltimore indicated that we can append an INFO after the TABLE tag to indicate overflow.]*

2.8 Versioning of the TAP protocol

The TAP protocol provides explicitly for versioning of the interface, using the features provided by the VOA registry and the conventions of the DAL-2 architecture.

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 is versioned separately and TAP and ADQL versions may differ.

Version numbers follow IVOA document conventions and contains two non-negative integers, separated by decimal points, in the form “x.y”, for example, “1.0”, or “1.13”. This is actually a three level version number encoded as two digits, e.g., “1.23” is logically the same as “1.2.3”. One result of this syntax is that second level version numbers cannot be greater than 9, for example “1.9” is a higher version number than “1.10” (logically “1.9.0 vs. “1.1.0”). Hence IVOA version numbers cannot be numerically compared without first being parsed.

2.8.2 Version number changes

The protocol version number will change with each published revision of this document. The number will increase monotonically and will comprise no more than two integers separated by decimal points, with the first integer being the most significant. There may be gaps in the numerical sequence. Some numbers may denote draft versions. Servers and their clients need not support all defined versions, but **must** obey the negotiation rules below.

A version number change at the first level (e.g., 1.0 – 2.0) indicates a major change. A version number change at the second level indicates a minor change which is not necessarily backwards compatible. A version number change at the third level is considered backwards compatible, and should not affect the pre-existing functionality of the interface.

2.8.3 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.

2.8.4 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. 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.

VOTables **should** comply with VOTable v1.1 or greater [9].

VOTables resulting from successful queries, including overflowed queries (see section Error: Reference source not found for a definition of overflow) **must** be returned with MIME type *text/xml;content=x-votable*. A base MIME-type of text/xml is used for synchronous queries to enable display of query results in browsers using direct rendering of the XML or an optional style sheet. VOTables which are manipulated as file data should instead use the MIME type *application/x-votable+xml*.

The VOTable **must** contain a *RESOURCE* element identified with the tag *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:

- “OK”, meaning that the query completed successfully and did not overflow;
- “ERROR”, meaning that an error was detected at the level of the TAP protocol;
- “OVERFLOW”, meaning that the query completed without error but overflowed;
- “STREAM”, meaning that neither error nor overflow had been detected when the service started to write the results to the client, but that either condition could still arise before the response is completed.

  [TAP 0.3 also says this concerning status reporting in streamed responses: 'Alternatively, the initial query status could be OK or ERROR and a failure later on would require just the additional INFO with OVERFLOW or ERROR - then we do not have to add STREAM... that might be a more general solution' – Ed.]

The STREAM status covers the case where the service streams a long table of results to the client rather than buffering it. In this situation, the data typically come from an SQL cursor and the service does not know the number of rows in the response when starting to write the *TABLE* element; overflow cannot be reported in the initial *INFO* element. When the initial status is set to STREAM, the service **must** write a second *INFO* element, with *name=”QUERY_STATUS”*, after the end of the *TABLE* element. This element **must** have its *value* attribute set to *“OK”, “ERROR”* or *“OVERFLOW”*.

The value of the *INFO* 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">Successful query</INFO>
<INFO name="QUERY_STATUS" value="ERROR">
  DEC out of range: DEC=91</INFO>
<INFO name="QUERY_STATUS" value="OVERFLOW">
Number of table rows exceeds default limit of 5000
</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:

```
 <INFO name="QUERY_STATUS" value="ERROR">unrecognized operation</INFO>
 <INFO name="SERVICE_PROTOCOL" value="1.0">TAP</INFO>
 <INFO name="REQUEST" value="doQuery"/>
 <INFO name="baseUrl" value="http://webtest.aoc.nrao.edu/ivoa-dal"/>
 <INFO name="serviceVersion" value="1.0"/
 <INFO name="serviceName" value="tap"/>
  <INFO  name="ServiceEngine"  value="tap:  TAP  1.0  DALServer  version
0.4"/>
```

2.9.2 Special Table Content

If the output of a query includes column(s) of type datetime/timestamp(?), the values **must** be specified in ISO8601 format.

If the output of a query includes columns of type region (e.g. a column of type *POINT, CIRCLE, POLYGON*, or *REGION* as defined by ADQL), the value **must** be output in a single column and encoded in STC-S format (Rots 2007). If the underlying tables (as described by the table metadata) store spatial information in multiple columns (e.g. RA and DEC in separate columns), then the output **may** also use multiple columns (and, in the case of VOTable, the coordinate system can usually be specified by a *PARAM* rather than a *FIELD*). In either case, the result should be presented in the same number of columns as were present in the SELECT (applicable to ADQL and PQL).

Where possible output table columns should be assigned UCDs (uniform content descriptors) to indicate the type of quantity stored in the column. If the table contains a data model columns may also be assigned UTYPEs, and may be

aggregated with the VOTable GROUP construct to identify a subset of table columns as a data model instance.

3 Service Registration (normative)

Publication of a service to the VO requires that it be registered with the VO 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. The content of this element, including the value of its *standardID* attribute is TBD.

[In the debate leading to TAP 0.3, it was suggested that the capability might list as interface the URL for the root web-resource of the service (as defined in section Error: Reference source not found). Clients would add to this URL /sync or /async as appropriate. This arrangement was not confirmed in the text of v0.3 and should be confirmed or replaced in TAP 0.4 – Ed.]

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].

[This requirement is not stated in TAP 0.3. I have added it since VOSI itself requires it – Ed.]

The resource document should include the table metadata, except where the database-schema of the archive changes frequently.[4] Where table metadata are provided, they **must** be represented as XML elements drawn from *VODataService* 1.1.

4 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

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 Error: Reference source not found . 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://x.y.z/TAP*. TAP mandates that the asynchronous requests be directed to *http://x.y.z/TAP/async*. This URL points to the list of 'jobs'; i.e. the list of queries currently or recently executed.

To start a new query, the client posts a request to the job list.

```
HTTP POST to http://x.y.z/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 *j42*, then the URL will be *http://x.y.z/TAP/async/j42* because the jobs are always children of the job list.

The service then issues an HTTP redirection to the job's URL.

```
HTTP status 303 'See other'
Location: http://x.y.z/TAP/async/j42
```

Beneath the job URL there are further URLs for aspects of the job:

```
http://x.y.z/TAP/async/j42/phase
http://x.y.z/TAP/async/j42/results
http://x.y.z/TAP/async/j42/error
```

(there are more, but these are the one that the client has to deal with).

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 a an error document explaining the problem. If the phase is *PENDING*, then the client needs to commit the job for execution.

The client commits the job by posting to the phase URL

```
HTTP POST to http://x.y.z/TAP/async/j42/phase
PHASE=RUN
```

The service replies with a redirection to the job URL

```
HTTP status 303 'see other'
Location: http://x.y.z/TAP/async/j42
```

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://x.y.z/TAP/async/j42/phase
```

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

```
HTTP GET http://x.y.z/TAP/async/j42/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.

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 URL to find out what went wrong

```
HTTP GET http://x.y.z/TAP/async/j42/error
```

The service remembers the job for a limited period after which it forgets the job information and discards the result of the query. After job expires, 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:

```
HTTP GET http://x.y.z/TAP/async/j42/destruction
```

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

```
HTTP POST to http://x.y.z/TAP/async/j42/destruction
DESTRUCTION=2008-11-11T11:11:11Z
```

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://x.y.z/TAP/async
 [read Location header from curl output]
curl -d 'PHASE=RUN' http://x.y.z/TAP/async/j42
curl http://x.y.z/TAP/async/j42/phase
[repeat until phase is COMPLETED]
curl http://x.y.z/TAP/j42/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. Ultimately one would like to have per-user VOSpace storage co-located with the TAP service, allowing user queries to save output tables to the local VOSpace as well as use them for input in subsequent queries, without having to serialize to and from VOSpace and transfer tables over the network. Frequently-used tables such as source lists for multi-position queries could persist between queries, and could be arbitrarily large.

The current version of TAP does provide limited VOSpace integration via the table *UPLOAD* parameter, using the upload URI to point to a table stored in either a local or remote 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 parameterized 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.

=	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]	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]) whenever a large amount of data needs to be uploaded inline in the query, e.g., when uploading an inline table, or whenever the request may change the server state, e.g., when requesting asynchronous execution of a query. Semantically POST and GET are largely the same, permitting the same parameters to be transmitted to the server to define the request. 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 Error: Reference source not found of this document, or send a service exception if unable to respond correctly. Only in the case of Version Negotiation (see 2.8.4) may the server offer a differing result. Upon receiving an invalid request, the server **must** issue a service exception as described in section Error: Reference source not found.

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 below. The basic structure of a MIME type is a string of the form “type/subtype”. MIME allows additional parameters in a string of the form “type/subtype; param1=value1; param2=value2”. A server may include parameterized MIME types in its list of supported output formats. In addition to any parameterized variants, the server should offer the basic unparameterized version of the format.

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

[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] G. Rixon & P. Harrison, *Universal Worker Service Version 0.5,* IVOA internal working-draft 8th October 2008. http://www.ivoa.net/internal/IVOA/AsynchronousHome/UWS-0.5.pdf

[4] A. Rots, *Space-Time Coordinate Metadata for the Virtual ObservatoryVersion 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] G. Rixon (ed.) & GWS-WG, *IVOA Support Interfaces Version 1.00*, IVOA Working Draft 2008 October 23. http://www.ivoa.net/Documents/WD/GWS/VOSI-20081023.pdf

[7] R, Plante, (ed.), A. Stébé, K. Benson, M. Graham, G. Greene, P. Harrison, A. Linde, G. Rixon & IVOA Registry-WG, *VODataService: a VOResource Schema Extension for Describing Collections and ServicesVersion 1.01.* IVOA Working Draft 16 October 2008. http://www.ivoa.net/internal/IVOA/VODataService/VODataService-v1.1wd.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 MetadataVersion 1.03*, IVOA Recommendation 22 February 2008. http://www.ivoa.net/Documents/REC/ReR/VOResource-20080222.html

[9] F. Ochsenbein (ed.), R. Williams, C. Davenhall, D. Durand, P. Fernique, D. Giaretta, R. Hanisch, T. McGlynn, A. Szalay, M. Taylor, A. Wicenec, *VOTable Format DefinitionVersion 1.1*, IVOA Recommendation 11 August 2004. http://www.ivoa.net/Documents/REC/VOTable/VOTable-20040811.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

[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