International

Virtual

Observatory

Alliance

Universal Worker Service
Version 0.4
IVOA Internal Working Draft 2008 June 05

This version:

0.4-2006-05-11

Latest version:

not issued outside GWS-WG

Previous version(s):

Internal Working Draft v0.1, 2005-01-24

Internal Working Draft v0.2, 2006-05-11

Internal Working Draft v.0.3, 2007-04-26

Author(s):

Guy Rixon, Paul Harrison

Abstract

The Universal Worker Service pattern (UWS) defines how to manage asynchronous execution of jobs on a service. Any application of the pattern defines a family of related services with a common service contract. Four possible applications are described.

Status of This Document

This is an internal working draft of the GWS-WG. The first release of this document was on 2005-01-24 within the working group; it has not yet been issued outside the working group.

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 made obsolete 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

The need for the UWS pattern was inspired by AstroGrid's Common Execution Architecture and particularly by discussions with Noel Winstanley. The ideas about statefulness are distilled from debates in the Global Grid Forum in respect of the Open Grid Services Infrastructure that was the fore-runner of Web Services Resource Framework. The REST binding came initially from suggestions by Norman Gray.

Contents

1 Introduction

The Universal Worker Service (UWS) pattern defines how to build *asynchronous*, *stateful*, *job-oriented* services (the italicized terms are defined in sub-sections of this introduction). It does so in a way that allows for wide-scale reuse of software and support from software toolkits.

Section 2 of this document describes the pattern and lists the aspects that are common to all its applications. Any such application would involve a service contract that embodies the pattern and fixes the issues left undefined in the pattern itself. The contract would include the XML schemata (XSD and WSDL) for the application. It is intended that each such contract cover a family of related applications, such that the implementations may be widely reused.

Section 3 outlines several possible applications of the pattern. These use-cases may be expanded into full IVOA standards that are siblings of the current document.

1.1 Synchronous, stateless services

Simple web services are *synchronous* and *stateless*. Synchronous means that the client waits for each request to be fulfilled; if the client disconnects from the service then the activity is abandoned. *Stateless* means that the service does not remember results of a previous activity (or, at least, the client cannot ask the service about them).

Synchronous, stateless services work well when two criteria apply.

The length of each activity is less than the “attention span” of the connection.

1. The results of each activity are compact enough to be easily passed back to the client via the connection on which the request was made (and possibly pushed back to the service as parameters of the next activity).

There are various limits to the attention span.

HTTP assumes that the start of a reply quickly follows its request, even if the body of the reply takes a long time to stream. If the service takes

too long to compute the results and to start the reply, then HTTP times out at the request is lost.

A client runs computer which will not stay on-line indefinitely.

A network with finite reliability will eventually break communications during an activity.

A service is sometimes shut down for maintenance.

Synchronous, stateless services, in short, do not scale well.

1.2 Some IVO activities that outgrow synchronous, stateless services

These cases are examples. They are not a complete list!

1. An ADQL [1] service gives access to a large object-catalogue. Most queries run in less than a minute, but some legitimate queries involve a full-table traverse and take hours or days. The service needs to run these special cases in a low-priority queue.
2. An object-finding service runs the Sextractor application on a list of images. Normally, the list is short and the request is quickly satisfied. Occasionally, a list of 10,000 images is sent in the expectation that the work will be finished over the weekend.
3. A cone-search [2] request on a rich catalogue raises 10,000,000 rows of results, but the client is connected via a slow link and cannot read all the results in a reasonable time. The client needs the service to send the results into storage over a faster link. This could mean sending them to VOSpace, or simply holding them temporarily until the user can retrieve them on a fast link.
4. An ADQL service allows users to save query results into new tables such that they can be the target of later queries. However, space is limited and the results tables can only be kept for a short time. The client and service negotiate the lifetime of the results tables.
5. A service performs image stacking on a list of fields. Each field can be processed by a synchronous service but the list is long and the user wants to retrieve the results of the early fields before the last fields are processed.

1.3 Asynchronous and stateful services

Services can be made to scale better by making them *asynchronous* and *stateful*. Asynchronous means that a client makes two or more separate requests to the service in the course of one activity, and that the client and service may be disconnected, possibly for days or more, in between those requests. Stateful means that the service stores state information about the activity and the client addresses requests to this state.

Web services that are asynchronous are almost always stateful. Most of special extra arrangements for asynchronous activities are actually managing the state of the activity.

There is an important class of stateful services where the state is peculiar to one job or session and the job is “owned” by one user. These, for the purpose of this document, are called *job-oriented* services. There are stateful services that are not job-oriented (e.g. a service managing a shared, client-writeable DB table), but UWS does not apply to these.

For the purpose of this discussion, let the term *job* refer to the work specified by the JDL instructions and the term *resource* refer to the state of the job as recorded by the service. Both have a finite duration. The *lifetime* of the resource – i.e. the time from inception until the service forgets the state – is generally finite and must be at least as long the duration of the job.

1.4 Job description language, service contracts and universality

Consider the web-service operation that starts off a job. This operation must express what is to be done in the activity: it must carry parameters in some form.

The parameters may be expressed as a list. E.g., a cone search service takes a list of three parameters: RA, DEC, RADIUS. Alternatively, the parameters may be arranged as an XML document (e.g. ADQL, CEA). The rules for setting and arranging the parameters for a job are called the *Job-Description Language* (JDL).

The combination of the UWS pattern, a JDL and details of the job state visible to the client defines a service contract; for a SOAP service, this contract can be captured in WSDL. Changing the JDL changes the contract. Thus, it is not meaningful to “implement UWS” in isolation; any implementation standard must specify the rest of the contract.

If the JDL is very general, a single service-contract can be reused for many kinds of service. AstroGrid’s CEA exploits this: one JDL covers all services offering parameterized applications and even ADQL services. In the limit, a sufficiently-general JDL turns a specialized worker service into a universal worker service.

2 Universal Worker Service pattern

2.1 Objects within a UWS

A UWS consists logically in a set of objects that may be read and written to in order to control jobs.

In a SOAP binding of UWS, these components are exposed as properties of the object that lives at the endpoint registered for the service. In a REST binding, the components are distinct web-resources each with its own URI.

The following sub-sections explain the semantics of the objects. The UML diagram shows the relationships more succinctly.

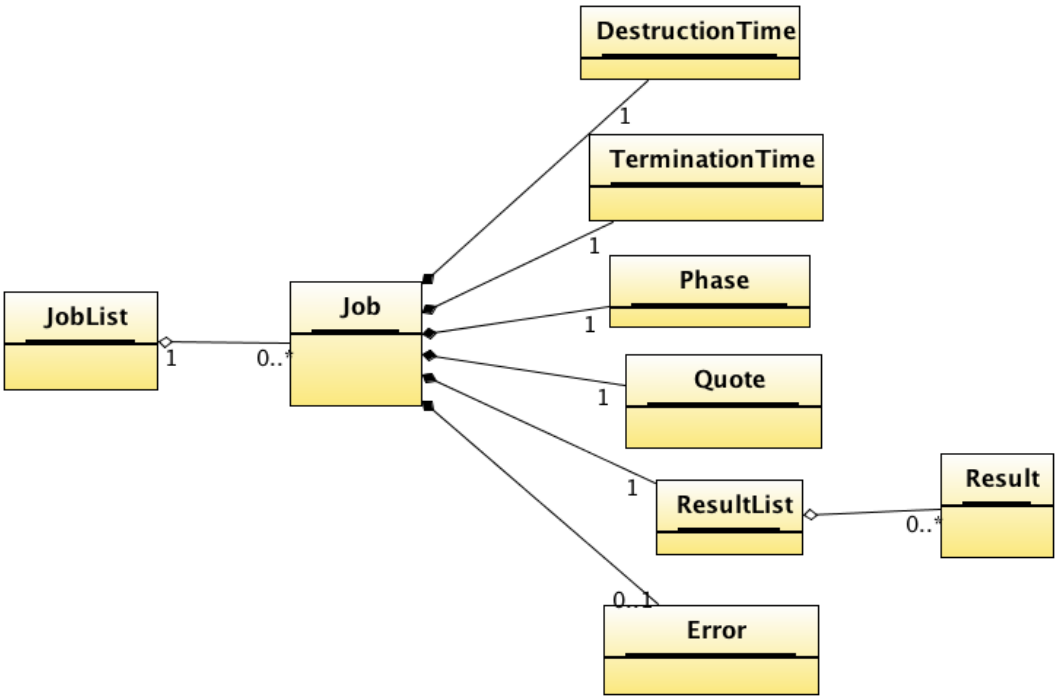


Illustration 1: UWS Objects

2.1.1 Job list

The Job List is the outermost object: it contains all the other objects in the UWS. The immediate children of the job list are Job objects (see next sub-section).

The job list may be read to find the extant jobs.

The job list may be updated to add a new job.

The job list itself does not allow jobs to be deleted. Instead, when a job is destroyed by an action on its job object, then the list updates itself accordingly.

2.1.2 Job

A Job object contains the state of one job. The state is a collection of other objects. Each Job aggregates

- Exactly one Execution Phase.
- Exactly one Termination Time.
- Exactly one Deletion Time
- Exactly one Quote.
- Exactly one Results List.

2.1.3 Execution Phase

The job is treated as a state machine with the Execution Phase naming the state. The phases are

- PENDING: the job is accepted by the service but not yet committed for execution by the client. In this state, the job quote can be read and

evaluated. This is the state into which a job enters when it is first created.

- QUEUED: the job is committed for execution by the client but the service has not yet assigned it to a processor. No Results are produced in this phase.
- EXECUTING: the job has been assigned to a processor. Results may be produced at any time during this phase.
- COMPLETED: the execution of the job is over. The Results may be collected.
- ERROR: the job failed to complete. No further work will be done nor Results produced. Results may be unavailable or available but invalid; either way the Results should not be trusted.
- ABORTED: the job has been manually aborted by the user, or the system has aborted the job due to lack of or overuse of resources.

2.1.4 Termination Time

A Termination Timeobject defines the duration for which a job shall run. This represents the “computation time” that a job is to be allowed, although because a specific measure of CPU time may not be available in all environments this duration is defined in real clock seconds. A termination time of 0 implies unlimited execution duration.

When the termination time duration has passed the service will automatically abort the job, which has the same effect as when a manual “Abort” is requested.

Specifically, when a job is aborted:

- if the job is still executing, the execution is aborted.
- any results of the job are retained.

When a job is created, the service sets the initial termination time. The client may write to a Termination Time to try to change the job's cpu time allocation. The service may forbid changes, or may set limits on the allowed termination time.

2.1.5 Destruction Time

The Destruction Time object represents the instant when the job shall be destroyed. The Destruction Time is an absolute time.

Destroying a job implies;

- if the job is still executing, the execution is aborted.
- any results from the job are destroyed and storage reclaimed.
- the service forgets that the job existed.

The Destruction Time may be viewed as a measure of the amount of time that a service is prepared to allocate storage for a job – typically this will be a longer duration that the amount of CPU time that a service would allocate.

When a job is created the service sets the initial Destruction Time. The client may write to the Destruction Time to try to change the life expectancy of the job. The service may forbid changes, or may set limits on the allowed destruction time.

2.1.6 Quote

A Quote object predicts when the job is likely to complete. The intention is that a client creates the same job on several services, compares the quotes and then accepts the best quote.

The client may write to a Quote to accept the quote and commit the parent job for execution.

Quoting for a computational job is notoriously difficult. A UWS implementation must always provide a quote object, in order that the two-phase committal of jobs be uniform across all UWS, but it may supply a “don't know” answer for the completion time.

2.1.7 Error

The error object gives a human readable error message (if any) for the underlying job.

2.1.8 Results List

The Results List object is a container for formal results of the job. Its children may be any objects resulting from the computation that may be fetched from the service when the job has completed.

Reading the Results List itself enumerates the available or expected result objects.

The children of the Results List may be read but not updated or deleted. The client may not add anything to the Results List.

2.2 Bindings

2.2.1 REST binding

2.2.1.1 Resources and URIs

In a REST (Representational State Transfer) binding of UWS, each of the objects defined above is available as a web resource with its own URI. These URIs must a hierarchy as follows:

/(jobs)	the Job List
/(jobs)/(job-id)	a Job
/(jobs)/(job-id)/phase	the Phase of (job-id)
/(jobs)/(job-id)/termination	the Termination Time of (job-id)
/(jobs)/(job-id)/destruction	the destruction time for (job-id)
/(jobs)/(job-id)/error	any error message associated with (job-id)
/(jobs)/(job-id)/quote	the Quote for (job-id)

/(jobs)/(job-id)/results the Results List for (job-id)

The service implementor is free to choose the names given in parentheses above; the other names are part of the UWS standard.

The URI for the Job List, in its absolute form is the root URI for the whole UWS. This URI should be given as the access URL in the UWS' registration.

2.2.1.2 Representations of resources

For each of the resources, HTTP GET fetches a representation.

The representation of the Job List is a list of links to extant jobs. The list may be empty if the UWS is idle.

The representation of a Job is a list of links to its Phase, Termination Time, Destruction Time, Error, Quote and Results List.

The representation of a Results List is a list of links to the resources representing the results. These resources may have any URI and any MIME type. A sensible default for their URIs is to make them children of /(jobs)/(job-id)/results, but this is not required. It may sometimes be easier for a service implementor to point to a resource on some web server separate from that running the UWS. Therefore, a client must always parse the Results List to find the results. Where a protocol applying UWS specifies standard results it must do so by naming those results; the names then appear in the Results List in addition to the URIs. Not all results need to be named; sometimes the meaning of the result is obvious from the context and the name is omitted.

HTTP allows multiple representations of a resource distinguished by their MIME types and selected by the HTTP headers of a GET request. UWS exploits this to support both web browsers and rich clients in the same tree of resources.

A UWS should return HTML or XHTML to clients that accept these types. These clients are assumed to be web browsers and the UWS is generating its own user interface. The HTML interface generated should allow full control of the UWS via the use of HTML forms and appropriate links.

Clients which are assumed to be part of remote applications that drive UWS without showing the details to their users should accept only application/xml. A UWS must therefore return XML representations of the resources.

The XML schemata for the lists of links, and for the Phase, Termination Time and Quote documents, are detailed in Appendix A of this specification. They do not vary between UWS installations.

2.2.1.3 State changing requests

Certain of the UWS' resources accept HTTP POST and DELETE messages to change the state of the service – This is the fundamental way that a client controls the execution of a job.

2.2.1.3.1 Creating a Job

POSTing a request to the Job List creates a new job (unless the service rejects the request). The response when a job is accepted must have code

303 “See other” and the Location header of the response must point to the created job.

This initial POST will in most cases carry parameters for the protocol that is using the UWS pattern, as detailed in section 3.

2.2.1.3.2 Deleting a Job

Sending a HTTP DELETE to a Job resource destroys that job, with the meaning noted in the definition of the Job object, above. No other resource of the UWS may be directly deleted by the client. The response to this request must have code 303 “See other” and the Location header of the response must point to the Job List.

Posting a request with a parameter ACTION=DELETE to the Job also destroys the job, the response being as for a deletion. This action supports web browsers which typically cannot send DELETE requests.

2.2.1.3.3 Changing the Destruction Time

The Destruction Time may be changed by POSTing to /(jobs)/(job-id)/destruction. In this case, the body of the posted request is of type *application/x-wwww-form-urlencoded* and contains the parameter named TIME whose value is the new termination time in ISO8601 format; i.e. this request is what an HTML form sends.

The response to this request must have code 303 “See other” and the Location header of the response must point to the /(jobs)/(job-id)/destruction URI so that the client receives the value that the service has actually set the Destruction Time to.

2.2.1.3.4 Changing the Termination Time

The Termination Time may be changed by POSTing to /(jobs)/(job-id)/termination. In this case, the body of the posted request is of type *application/x-wwww-form-urlencoded* and contains the parameter named DURATION whose value is the new termination time in seconds.

The response to this request must have code 303 “See other” and the Location header of the response must point to the /(jobs)/(job-id)/termination URI so that the client receives the value that the service has actually set the Termination Time to.

2.2.1.3.5 Starting a Job

A job may be started by POSTing to the /(jobs)/(job-id)/phase URI. The POST contains a single parameter PHASE=RUN which instructs the UWS to attempt to start the job. The response to this request must have code 303 “See other” and the Location header of the response must point to the /(jobs)/(job-id)/phase URI so that the client receives the phase that the job has been set to. Typically a UWS will put a job into a QUEUED state on receipt of this command, but depending on how busy the server is, the job might be put almost immediately into an EXECUTING state.

2.2.1.3.6 Aborting a Job

A job may be aboted by POSTing to the /(jobs)/(job-id)/phase URI. The POST contains a single parameter PHASE=ABORT which instructs the UWS to attempt to abort the job. The response to this request must have code 303 “See other” and the Location header of the response must point to the /(jobs)/(job-id)/phase URI so that the client receives the phase that the job has been set to.

2.2.1.4 Message pattern

The REST binding results in the message pattern shown in figure 2.

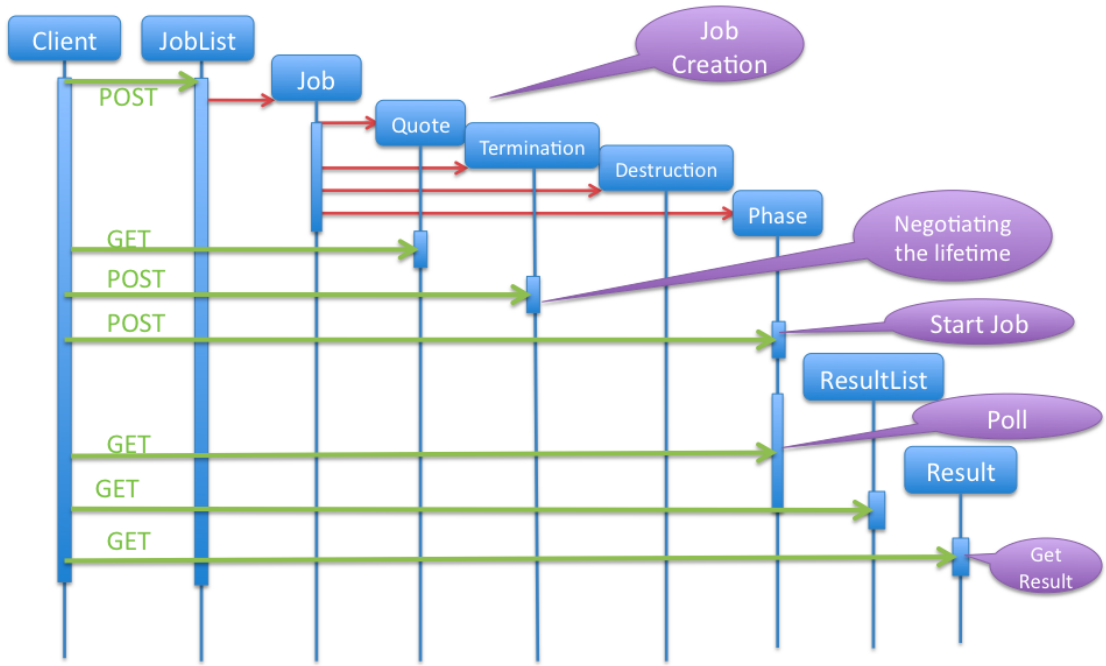


Illustration 2: Typical calling sequence for the REST binding of UWS

2.2.2 SOAP binding

2.2.2.1 Resources and URIs

For a SOAP-bound UWS there is only one web resource and therefore one URI. This resource corresponds roughly to the Job List object of the UWS and all the other objects are accessed via methods on the Job List.

The single URI of a SOAP-bound UWS must be registered as its access URL.

Since the SOAP binding has only a single URI it needs a different way to indicate the job to which a request applies. Job-specific requests and reponses must carry a SOAP header containing a WS-Addressing structure. In this structure, the *ResourceIdentifier* element names the resource; it is an opaque string to the client and is meaningful only to the service. The identifier for a given job is stated in the response to the request that creates it; that request is one that does not need a WS-Addressing header.

The exact use of WS-Addressing will be stated in the WSDL for the SOAP binding which is TDB.

2.2.2.2 Representations of objects

The SOAP binding allows the client to retrieve representations of some of the UWS objects. Unlike the REST binding, it does not allow all the objects to be represented. Further, the SOAP binding only provides XML representations; it does not support HTML in any way.

Representations may be got from the following methods.

- getJobs()
- getTerminationTime()
- getDestructionTime()
- getQuote()
- getPhase()
- getResults()
- getError()

Note that there is no representation of a Job as a whole. The method getResults() packs all the results in one XML document and returns that document. The details of the available results are defined by applications of the UWS pattern.

2.2.2.3 State-changing operations

The following methods of a UWS change its state; most of them change the state of one particular job.

- createJob()
- setTerminationTime()
- setDestructionTime()
- execute()
- abort()
- destroy()

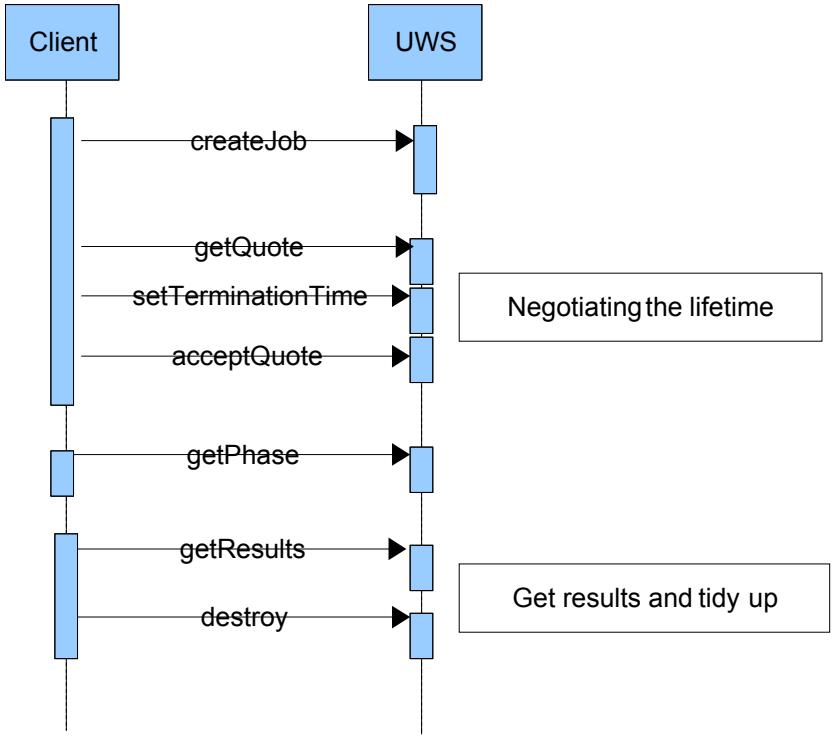
The excute() method commits a job for execution. The destroy() method destroys the target job.

The exact content posted to create a job is defined by applications of the UWS pattern.

2.2.2.4 Message pattern

The SOAP binding has the message pattern shown in figure 3.

Note that the number of objects is much lower than in the REST binding and the number of messages lower by one. However, the complexity of each message is significantly greater. The overall complexity of the pattern is roughly the same in each binding.


Client
UWS
createJob
getQuote
setTerminationTime
acceptQuote
Negotiating the lifetime
getPhase
getResults
destroy
Get results and tidy up

3 Applications of UWS

The UWS pattern leaves undefined two essential parts of the service contract: the content that must be posted to create a job; and the pattern of results made available by a completed job. An application of UWS completes a service contract by defining these matters.

There follow some use cases applying the UWS pattern. The descriptions are neither formal nor complete. The intention is to show a range of ways that the pattern can be applied without burdening the reader with the level of detail needed for a standard implementation.

These applications are carried over from v0.2 of this specification and hence they use the SOAP binding. It is intended to change the examples to REST binding in a later draft of this standard.

Any of these cases could be worked up into a full IVOA standard by formalizing the description, adding detail (schemata, WSDL) and generally making the specification more exact and complete. I suggest that each case so treated be broken out into a separate specification-document.

3.1 Image service with data staging

Consider a service that computes images from archive data. The computation takes significant time, so the service is asynchronous. The service keeps the computed images in its own storage until the user downloads them; this is essentially the model of SIAP [9].

The asynchronous image-service is a logical extension of a synchronous SIA service. Therefore it uses the REST binding of UWS.

The parameters for posting a new job are as for SIAP 1.0:

- POS, the position on the sky to be searched;
- SIZE, the size of the search box;
- FORMAT: the type of images to be computed.

Particular implementations are free to add extra parameters.

These parameters are posted in a document of type *application/x-www-form-urlencoded*: i.e. they can be sent from an HTML form.

The images generated by the job are accessible as unnamed results. Each image has its own URI and can be downloaded over HTTP at any time until the termination time of the job. The URIs for the images may be discovered from the Results List in the normal UWS way.

SIAP 1.0 produces, for each query, a table of metadata describing the images. The asynchronous image-service produces a table to the same schema as a named result, called “table”.

Image results are added to the results list, and to the “table” result, as they are generated. Hence, a client that polls the service can discover, download and use some of the images before the job is finished. If the

client is satisfied with these early images, the client can cancel the rest of the job by destroying the job. However, destroying the job deletes the cached images so the client has to download them *first*.

3.2 ADQL service with cursor

ADQL [1] can serve as a JDL. Consider an ADQL service that supports long-running queries as asynchronous operations. In general, the results of the query may be a large set of data. They may be too large to download comfortably. We might like to cache these results on the service and to operate a cursor, drawing down from the resource a few rows of the table at a time.

The parameters of a job are as follows:

- ADQL: the query text
- FORMAT: the format for the results

These parameters are posted in a document of type *application/x-www-form-urlencoded*: i.e. they can be sent from an HTML form.

A successful query generates the following, named results.

- *table*: the whole result set as one file resource.
- *header*: the metadata for the output table.
- *cursor*: a selection of rows of output.

The *cursor* result is parameterized by the query parameters FIRST and LAST in the query string of its URI: these parameters state the index of the first and last row to be returned; e.g.

> http://whatever.org/adlqService/results/cursor?FIRST=1&LAST=100

If the parameters are missing, the service decides which rows to emit.

3.3 VOSpace with controlled lifetime

The VOSpace [10] standard describes how a distributed network of storage can be built up from individual VOSpace services. It is intended that much of this storage be short-term scratch space, that some be available for a longer period, and that only a tiny part be permanent. Most VOSpaces have finite lifetimes.

VOSpace itself does not address how the lifetime of a particular space is determined, controlled, enforced or communicated to a user. This could lead to confusion, and may involve the operators of a space in much work when seeking to reclaim storage. The UWS pattern can make the lifetime of the space explicit.

Consider a VOSpace service that creates spaces on demand for authorized users. “Creating a space” means that the service creates VOSpace container-node (a kind of virtual directory), assigns a VOSpace identifier for it, records internally that the container belongs to the requesting user (such that only that user can create, modify or delete data-nodes and container nodes within the initial container) and *sets a finite lifetime for the container*.

This is a job-oriented service. The job is the created space and the JDL specifies how much storage and the desired lifetime. The management can be done with the UWS pattern, but with one special addition: the job itself – i.e. the space – is exposed to the client.

VOSpace 1 is a SOAP protocol. Therefore, the lifetime controls for the space uses the SOAP binding of UWS.

A space (job) is created by a posted SOAP-request defining the terms of the requested lease. The details of this request are better captured by an XML document, with schema, than by an RPC: the document is more capable of later expansion and specialization. For easier interoperation, the service should used the “wrapped” style of document/literal SOAP; i.e., the request document is wrapped in an XML element named for the request operation (“lease”, say, but the name is not critical) and that element is in turn wrapped in the SOAP envelope.

Since the job is to create the VOSpace, it completes quickly. It has one, named result, “space”, whose value is the VOSpace identifier of the created space. The termination time of the job is the instant at which the VOSpace will be revoked and its contents destroyed. The client can negotiate for lease extensions in the normal, UWS way. In particular, if the user is finished with the storage before the lease expires, he can explicitly give it back.

The URI for the *space* result is the *vos://* identifier for the space itself. This is an exceptional use of UWS where the result of the job is not something that can be downloaded. To find out the identifier, the client calls *getResults()* and receives a SOAP envelope containing one XML element, e.g.

```
<xsd:anyUri>vos://astrogrid.org!VOSpace-2/foo/bar</xsd:anyUri>
```

3.4 Parameterized applications

There is a class of applications on which a job may be defined by a list of simple parameters. “Simple” here means unstructured: a scalar value such as a number, a string of text or a Boolean value. If the parameters are allowed to name files, so that structured data are passed indirectly, then the class of applications is very large indeed: almost any non-interactive application can be driven in this way.

Turning each application of choice into a service (with or without UWS semantics) would be onerous. However, if the application’s interface is entirely characterized, through the JDL, in terms of typed input and output parameters, then one service contract will work for all the applications and a single implementation of the contract can be reused for all cases.

AstroGrid’s Common Execution Architecture (CEA) [11] works in this way. It has just one service contract for all applications (including ADQL services; the ADQL query is passed in the list of parameters). It has four implementations, one for each of the possible interfaces between the service and a kind of job (jobs can be implemented with Java classes, command-line applications, HTTP-get services or JDBC databases). CEA also specifies stateful, asynchronous services and makes use of VOSpace.

Consider a CEA reworked to use the UWS pattern for consistency with other (future) IVOA standards. Call it CEA v2 to distinguish it from CEA v1 as currently maintained by AstroGrid. For this example, consider the particular kind of CEA service that runs applications supplied as executable binaries.

A binary application-server has a library of applications co-located with its service and defined in the service configuration set by the service provider. It does not accept code from the client for local execution.

The JDL in CEA v2 is similar to that in CEA v1 [11]. It is a formal, XML vocabulary for expressing choice of application and parameter lists [12]. Parameters may be inputs or outputs of the job.

To start a job, a document in this JDL is posted to the UWS. The document is sent in its native MIME-type, application/xml, so this is not an interface that can be driven directly from an HTML form.

The results of the job depend on the choice of application. They are all named results and the names and types are defined in the definition of the application. That application-definition is registered, so the client knows before running the job what results to expect.

CEA input-parameters may be indirect: i.e. they may refer to data in on-line storage. In this case, the JDL document contains the URIs for those data objects Alternatively, the parameters may be direct, in which case the JDL contains the actual value of the parameters.

Similarly, CEA results may be made indirect. In this case, the results are named as parameters in the JDL where the values are the URIs to which the results are delivered. The application server can then stream the results to the specified destination as they become available and need not cache them locally. If a job result is indirect, then the server can choose whether or not to keep a local copy. If it chooses not to cache locally, then the result URI in the UWS is set to the external location named in the URI.

4 References

[1] M. Ohishi, A. Szalay (eds.), IVOA Astronomical Data query Language, http://www.ivoa.net/Documents/latest/ADQL.html

[2] US NVO project, *NVO compliance: Simple Cone Search*, http://us-vo.org/pubs/files/conesearch.html

[3] D. Box, F. Curbera (eds.), *Web Services Addressing (WS-Addressing)*, http://www.w3.org/Submission/ws-addressing/

[4] T. Banks (ed.), *Web Service Resource Framework (WSRF) – Primer*, http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-01.pdf

[5] S. Graham, A. Karmarkar, J. Mischkinsky, I. Robinson, I. Sedukhin (eds.), *Web Services Resource 1.2 (WS-Resource)*, http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf

[6] S. Graham,, J. Treadwell (eds.), *Web Services Resource Properties 1.2 (WS-ResourceProperties)*, http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-os.pdf

[7] L. Srinivasan, T. Banks (eds.), *Web Services Resource Lifetime 1.2 (WS-ResourceLifetime)*, http://docs.oasis-open.org/wsrf/wsrf-ws_resource_lifetime-1.2-spec-os.pdf

[8] S. Graham, D. Hull, B. Murray, *Web Services Base Notification 1.3*, http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-pr-02.pdf

[9] D. Tody, R. Plante, *Simple Image Access Specification*, http://www.ivoa.net/Documents/latest/SIA.html

[10] Grid and Web Services Working Group of IVOA, work in progress, http://www.ivoa.net/twiki/bin/view/IVOA/IvoaGridAndWebServices

[11] P. Harrison, *Proposal for a Common Execution Architecture*, http://www.ivoa.net/Documents/latest/CEA.html

[12] P. Harrison, XML schema for namespace *http://www.astrogrid.org/schema/CommonExecutionArchitectureBase/v1*, http://software.astrogrid.org/schema/cea/CommonExecutionArchitectureBase/v1.0/CommonExecutionArchitectureBase.xsd

1 Appendix A Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- $Id: UWS.xsd,v 1.1.2.2 2008/05/07 12:15:00 pah Exp $ -->
<!-- proposal for basic UWS schema - Paul Harrison May 2008 -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.ivoa.net/xml/UWS/v0.9"
xmlns:uws="http://www.ivoa.net/xml/UWS/v0.9"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   elementFormDefault="qualified"
   >
   <xs:import namespace="http://www.w3.org/1999/xlink"
schemaLocation="../../../stc/STC/v1.30/XLINK.xsd"/>
<!--
  <xs:import namespace="http://www.w3.org/1999/xlink"
schemaLocation="http://www.ivoa.net/xml/Xlink/xlink.xsd"/>
-->

   <xs:complexType name="ShortJobDescription">
      <xs:sequence>
         <xs:element ref="uws:phase"></xs:element>
      </xs:sequence>
      <xs:attribute name="id" type="uws:job-identifier-type"
use="required"></xs:attribute>
      <xs:attributeGroup ref="uws:reference"></xs:attributeGroup>
   </xs:complexType>
   <xs:attributeGroup name="reference">
    <xs:annotation>
      <xs:documentation>standard xlink references</xs:documentation>
    </xs:annotation>
    <xs:attribute ref="xlink:type" use="optional" default="simple"/>
    <xs:attribute ref="xlink:href" use="optional"/>
  </xs:attributeGroup>

  <xs:simpleType name="ExecutionPhase">
```

```
      <!-- need to think a little here about the implication of allowing a
"re-entrant" application that is capable of running mini-jobs...probably
this is indicated with a different state varible entirely -->
      <xs:annotation>
         <xs:documentation>
            Enumeration of possible phases of job execution
         </xs:documentation>
      </xs:annotation>
      <xs:restriction base="xs:string">
         <xs:enumeration value="PENDING">
            <xs:annotation>
               <xs:documentation>
                  The first phase a job is entered into - this is where a
job is being set up but no request to run has occurred.
               </xs:documentation>
            </xs:annotation>
         </xs:enumeration>
         <xs:enumeration value="QUEUED">
            <xs:annotation>
               <xs:documentation>
                  An job has been accepted for execution but is waiting
                  in a queue
               </xs:documentation>
            </xs:annotation>
         </xs:enumeration>
         <xs:enumeration value="EXECUTING">
            <xs:annotation>
               <xs:documentation>An job is running</xs:documentation>
            </xs:annotation>
         </xs:enumeration>
         <xs:enumeration value="COMPLETED">
            <xs:annotation>
               <xs:documentation>
                  An job has completed successfully
               </xs:documentation>
            </xs:annotation>
         </xs:enumeration>
         <xs:enumeration value="ERROR">
            <xs:annotation>
               <xs:documentation>
                  Some form of error has occurred
               </xs:documentation>
            </xs:annotation>
         </xs:enumeration>
         <xs:enumeration value="UNKNOWN">
            <xs:annotation>
               <xs:documentation>
                  The job is in an unknown state
               </xs:documentation>
            </xs:annotation>
         </xs:enumeration>
         <xs:enumeration value="HELD">
            <xs:annotation>
               <xs:documentation>
                  The job is HELD pending execution and will not
                  automatically be executed (cf pending)
```

```
               </xs:documentation>
            </xs:annotation>
         </xs:enumeration>
         <xs:enumeration value="SUSPENDED">
            <xs:annotation>
               <xs:documentation>
                  The job has been suspended by the system during
                  execution
               </xs:documentation>
            </xs:annotation>
         </xs:enumeration>
         <xs:enumeration value="ABORTED">
            <xs:annotation>
               <xs:documentation>
                  The job has been aborted, either by user request or by the
server because of lack or overuse of resources.
               </xs:documentation>
            </xs:annotation>
         </xs:enumeration>
      </xs:restriction>
   </xs:simpleType>

   <xs:complexType name="JobSummary">
      <xs:sequence>
         <xs:element name="jobId" type="uws:job-identifier-type" />
         <xs:element ref="uws:phase" />
         <xs:element ref="uws:quote" />
         <xs:element name="startTime" type="xs:dateTime" />
         <xs:element name="endTime" type="xs:dateTime" />
         <xs:element ref="uws:termination" />
         <xs:element ref="uws:destruction"/>
     </xs:sequence>
   </xs:complexType>
   <xs:simpleType name="job-identifier-type">
      <xs:annotation>
         <xs:documentation>
            The identifier for the job
         </xs:documentation>
      </xs:annotation>
      <xs:restriction base="xs:string" />
   </xs:simpleType>

   <xs:element name="job" type="uws:JobSummary">
      <xs:annotation>
         <xs:documentation>
            This is the information that is returned when a GET is made
            for a single job resource - i.e. /(jobs)/(jobid)
         </xs:documentation>
      </xs:annotation></xs:element>
   <xs:element name="phase" type="uws:ExecutionPhase">
      <xs:annotation>
         <xs:documentation>
            the execution phase - returned at /(jobs)/(jobid)/phase
         </xs:documentation>
      </xs:annotation></xs:element>
   <xs:element name="quote" type="xs:dateTime">
```

```
      <xs:annotation>
         <xs:documentation>
            A Quote predicts when the job is likely to complete - returned
at /(jobs)/(jobid)/quote
            TODO - how to encode "don't know"
         </xs:documentation>
      </xs:annotation></xs:element>
   <xs:element name="termination" type="xs:dateTime">
      <xs:annotation>
         <xs:documentation>
            The time at which the job should be aborted if it is still
            running - returned at /(jobs)/(jobid)/termination
         </xs:documentation>
      </xs:annotation></xs:element>
   <xs:element name="destruction" type="xs:dateTime">
      <xs:annotation>
         <xs:documentation>
            The time at which the whole job + records + results will be
destroyed. returned at /(jobs)/(jobid)/destruction
         </xs:documentation>
      </xs:annotation>
      </xs:element>


   <xs:element name="jobList">
        <xs:annotation>
           <xs:documentation>
              The list of job references returned at /(jobs)
           </xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:annotation>
               <xs:documentation>
                  ISSUE - do we want to have any sort of paging or
                  selection mechanism in case the job list gets very
                  large? Or is that an unnecessary complication...
               </xs:documentation>
            </xs:annotation>
            <xs:sequence>
            <xs:element name="jobref" type="uws:ShortJobDescription"
maxOccurs="unbounded" minOccurs="0"></xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>


   <xs:complexType name="ResultReference">
      <xs:annotation>
         <xs:documentation>
            A reference to a UWS result
         </xs:documentation>
      </xs:annotation>
      <xs:sequence>
         <xs:element ref="uws:phase"></xs:element>
      </xs:sequence>
      <xs:attribute name="id" type="xs:string"></xs:attribute>
```

```
      <xs:attributeGroup ref="uws:reference"></xs:attributeGroup>
   </xs:complexType>
   <xs:element name="resultList">
        <xs:annotation>
           <xs:documentation>
              The element returned for /(jobs)/(jobid)/results
           </xs:documentation>
        </xs:annotation>
        <xs:complexType>
         <xs:sequence>
            <xs:element name="result" type="uws:ResultReference"
maxOccurs="unbounded" minOccurs="0"></xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

2 Appendix B WSDL 2.0 for REST binding

WSDL 2.0 has enhanced functionality for specifying REST web services, and so this

TBD

3 Appendix C WSDL 1.0 for SOAP binding

As there are more tools that work with the existing WSDL 1.0 specification, the SOAP binding is given in WSDL 1.0