Notes on Asynchronous Access Services

Authors:

Stefano Nativi, Paolo Mazzetti	Italian National Research Council (CNR-IMAA)
Ethan Davis	UNIDATA/UCAR

Contributors:

Document type: Discussion Paper

Distribution: Public

Status: Draft

Date: 23 November 2007

Objective

To discuss a general framework for asynchronous interaction with geospatial Web Services
To establish a first draft for a discussion paper to be submitted to the OGC

Scope

Grid Computing support of Access Services (e.g. OWS)
Support of long-running processing
Support of redirection and “shared access”.

Introduction

Several data access use-cases require to support asynchronous interaction in order to avoid to maintain connections open. In particular this is true for:

- long-running processes creating dynamic resources accessed as features, coverages or maps;
- interaction with asynchronous infrastractures (e.g. grid);

Moreover, to avoid multiple run, a storage functionality could be useful to allow multiple or shared access to the process output. In this context, the "storage" term implies keeping resources for some amount of time: the time period a resource is stored might be "persistent" or "long-term", that it doesn't go away after the first access.

Finally, a push interaction mode would reduce the number of requests allowing to implement publish/subscribe model for event notification in a complete asynchronous framework.

Architectural note

Data Access OWS typically allow two different bindings:

1. SOAP
2. POX-HTTP (Plain-Old-XML over HTTP)

SOAP binding relates to the W3C Web Services architecture, a Service-Oriented architecture built on top of the Web Architecture using SOAP information model and encoding [WS-ARCH].

POX-HTTP binding defines an encoding of the Service-Oriented architecture using common Web technologies. It relates to the W3C Web Architecture, a Resource-Oriented architecture based on the REST architectural style [WEB-ARCH]. While OWS POX binding is not a RESTful implementation it is still implicitly resource-oriented since it is based on a uniform interface (get*Something*) on different resources (capabilities, descriptions, coverages, features, etc.).

The choice of one of these different architectures is out of the scope of this document. Anyway all the details are provided referring to the POX-HTTP approach. A complete RESTful implementation framework is under development by the OGC REST Sub-Committee.

Functional requirements

We propose to extend data access services to support:

F 1. Asynchronous access. The response data are not provided in the response message. They can be retrieved later on the same or different URI. This functionality is negotiated. It is server-initiated (server decides that it cannot provide the response in a "short" time). (If the client needs an asynchronous data access it should start a new computing task handling the request).

F 2. Storage. The response is available on a different (stable or temporary) URI for multiple or shared access. The new URI validity is negotiated between client and server. The client can require a storage expiration time; the server decides the expiration time.

F 3. Push mode. The requestor is informed when the response is ready.

Implementation

Asynchronous Access

The Asyncronous Access is decided by the server. The Asynchronous Access support must be reported in the server capabilities.

In case of Asynchronous Access the server answers with a redirection message containing a status monitor or a reference to it. The status monitor includes one or more of the following fields:

a) the URI where the response is (or will be) available;
b) the estimated time for response completion;
c) the progress in the response completion;

It conforms to the ExecuteResponse defined in the OGC WPS Specification [OGC-WPS], with the following assumption:

Identifier is an unique identifier built using request parameters. E.g. it could be the URI used in the KVP encoding.

Note that StatusLocation use ('*Include when "store" is true in request*') is still valid. When a RESTful binding is used, the StatusLocation information should be reported in the Location header field to allow automatic redirection.

Storage

The Storage is requested by the client using a *store* parameter.

```
store ::= true | false
```

The default value is False.

The client can provide an expiration time using a *expirationTime* parameter.

```
expirationTime ::= <Date>
```

The storage expiration time is provided by the server in the response basing on the client's request and server's capabilities.

'Push' Interaction

The push interaction is requested by the client using a *callback* parameter.

```
callback ::= <URI>
```

The *callback* parameter is the URI where event notification (e.g. response completion) should be sent.

The 'push' functionality implementation is post-poned.

Use cases

The following table covers the main use cases. For simplicity reasons, almost all of the present access service implementations achieve pull retrieves without redirection.

	NO STORAGE		STORAGE	
	Pull	Push	Pull	Push
SYNCHRONOUS INTERACTION	store = False; *(the WCS 1.0 case)*	▪ store = False; ▪ call-back parameter must be provided by the client ▪ Server sends the resource content, as soon as possible	▪ store = True; ▪ Server answers with a redirection message	▪ store = True; ▪ call-back parameter must be provided by the client ▪ Server sends the resource address, as soon as the content is ready
ASYNCHRONOUS INTERACTION	▪ store = False; ▪ Client must re-issue the same GetCoverage request until the server is able to provide back the coverage	▪ store = False; ▪ call-back parameter must be provided by the client ▪ Server sends the resource content, as soon as possible	▪ store = True; ▪ Client must retrieve the resource status information in a polling way	▪ store = True; ▪ call-back parameter must be provided by the client ▪ Server sends the resource address, as soon as the content is ready

Synchronous Interactions

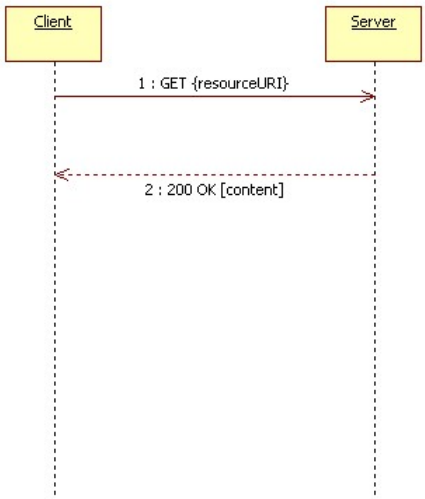
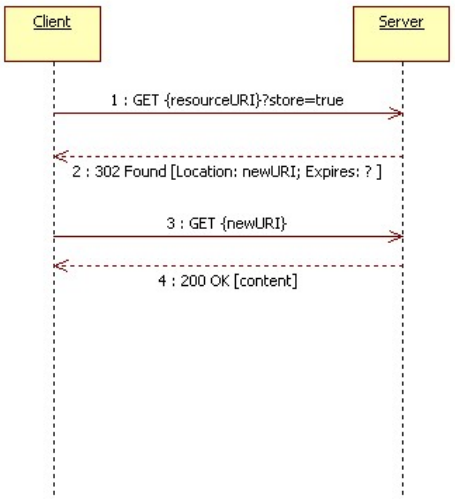
Asynchronous Interactions

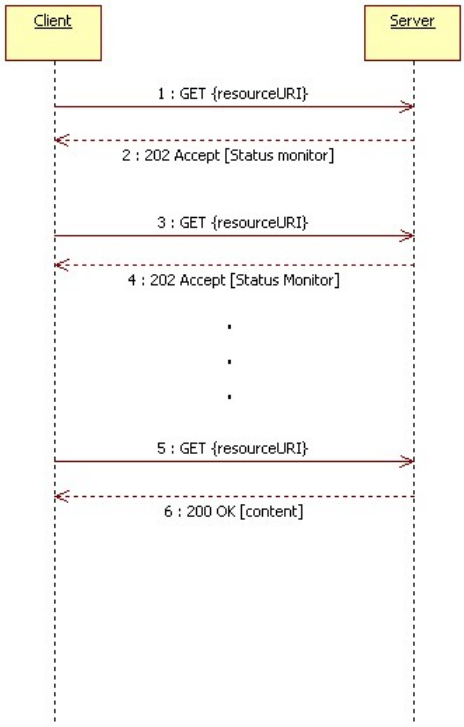
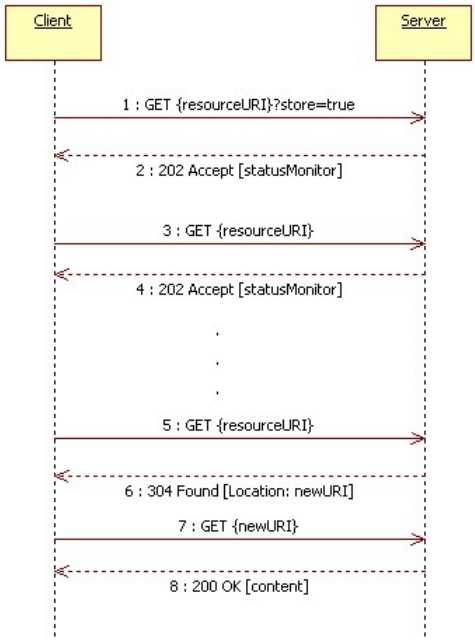
(*) Push interactions are shown in gray since they are not fully discussed in this paper

Sequence diagrams

The following figures show the sequence diagrams for different interactions with different *storage* capabilities (i.e. *storage* or no *storage* capability) and interaction modes (i.e. *synchronous* or *asynchronous*).

The sequences are presented using HTTP request and response messages with KVP parameters encoding in a RESTful style. This is only for presentation purposes.

		
No-Storage capability **Synchronous access** **Pull** retrieving		**Storage** capability **Synchronous access** **Pull** retrieving

		
No-storage capability **Asynchronous access** **Pull** retrieving		**Storage** capability **Asynchronous access** **Pull** retrieving

Notes on RESTful binding

This section of the document has to be considered informative. It collects several issues from the discussion in the WCS+ mailing-list. We considered useful to report it in this discussion paper both for reference and future enhancements.

Architectural issues

Generally speaking, great attention must be payed to the complexities that the architectural choice imposes on the design of clients. Indeed every little bit of extra complexity required of a client would drastically reduce the number of clients that get developed. One server, many clients: keep the client simple. REST architectures are characterized by uniform interfaces for accessing resources. In the most common implementations (e.g. over the HTTP protocol) only a very small set of actions is allowed (typically matching the CRUD pattern). Hence a part of the business logic must be moved from the server to the accessing nodes (e.g. clients). This means that in a REST architecture we need to build specific clients applications (e.g. using hyperlink navigation, or specific mobile code loaded on top of a light client like a browser implementing only access to the uniform interface). On the other side in a SOA infrastructure we can have generic but complex clients (i.e. capable of handling registry queries, interpreting service descriptions, calling services and so on).

Resources and representations

The REST architectural style is resource-oriented. It main concerns the identification of resources and the transfer of their representations.
In the OWS domain different subsets, interpolation, etc. identify different resources and not simply different representations.

a) In the KVP request encoding (GET) the query string parameters are not the set of input parameters for a single processing service resource, but actually parts of different resources identifiers. (Indeed only the parameter FORMAT should be considered affecting the representation and not identifying the resource. In a perfect REST world its content should be provided in the Accept header field.).

b) In the XML request encoding (POST) the target resource is a factory resource providing representations of children which are not individually identified. The requested child is specified by the XML encoded parameters provided in request message.

In the asynchronous access, what is provided by the possible redirection is not a new resource but a (temporary) URI to access it.

In the GET-KVP case it is an alias of the original URI for the same resource (a resource can have more than an URI). For example the resource `http://someserver.net/coverages/foo?bbox=...` is

assigned a temporary identifier `http://someserver.net/coverages/temp/xyz`. Anyway the resource is still retrievable at the original (and authoritative URI). This alias is useful because, for example, in the time range of its validity the retrieving of the resource representation could be faster than the retrieving from the original (canonical) URI.

In the POST-XML case the generated URI individually identifies the resource. To maintain the URI persistance it could be useful to generate URIs which are univocally dependant on the provided parameters. (The corresponding KVP encoding could be the starting point).

Caching

The REST architectural style has proven to be scalable, but it performs better for mostly-read applications thanks to the adoption of the multi-layer caching.
In an OWS data access service the cache management poses several issues:

a) In the XML encoding (POST) the cache should maintain the relationship between the XML-encoded parameters (message body) and the returned representation. This can be difficult due to the fact that different XML documents can have the same semantics.
b) In the KVP-encoding (GET) the cache should maintain the relationship between the URI (and header fields values) and the returned representation. Anyway it is not easy to recognize that two requests are the same, in particular due to the query string which is made of non-hierarchical parameters. (E.g. two requests could only differ for the parameters order.). While some hierarchical parameters (e.g. *name*) can be moved from the query string to the URI path, others are intrinsically non-hierarchical (e.g. *bbox*) and must remain part of the query string.
c) The same resource could be available in different formats. It would be useful to support resource caching with format transformation to repeat the extraction process for requests differing only for format.

This problem can be solved considering smarter cache managers with advanced functionalities:

1. Request canonicalization. MD5 hash of the 'canonicalised' request & using this as the key in a key-value-pair map; i.e. you use the request hash to look up a previous response (if any). This is how standard web-proxies. Issues are (1) how many requests to store, & (2) how do you know when the cache expires.
2. Request identification. Convert the query string into a low-level set of data extraction parameters (i.e. the parameters that are passed to NetCDF libraries for example, to extract a block of data) and cache these low-level parameters instead. These parameters typically consist of a file name, internal variable id and a set of indices for each axis in the data file. Your system will then parse the query string into these low-level parameters and check for identical parameters in the cache.
3. Raw data caching. To allow people to download the same data in different formats without doing the extraction twice.

References

[OGC-WPS]
OGC, “OpenGIS® Web Processing Service”, 2005

http://portal.opengeospatial.org/files/?artifact_id=13149&version=1&format=pdf
[WEB-ARCH]
W3C, “Architecture of the World Wide Web, Volume One”, 2004, http://www.w3.org/TR/webarch/
[WCSPLUS]
WCS+ Mailing-list, http://www.unidata.ucar.edu/mailing_lists/archives/wcsplus/
[WS-ARCH]
W3C, “Web Services Architecture”, 2004, http://www.w3.org/TR/ws-arch/