Extreme Markup Languages

Scalable, document-centric addressing of semantic stores using the XPointer Framework and the REST architectural style.

Bryan Thompson
Systems Architect
SAIC
Advanced Systems and Concepts
3811 N. Fairfax Dr., Suite 850
Arlington VA 22203
USA
thompsonbry@saic.com

Graham Moore
empolis, UK
Unit B, Dorcan Complex
Swindon Wiltshire SN3 5HQ
United Kingdom
graham.moore@empolis.uk

Bijan Parsia
The University of Maryland, College Park
Maryland Information and Network Dynamics Laboratory, Semantic Web Agents Group
8400 Baltimore Ave., Suite 200
College Park MD 20742
USA
bparsia@isr.umd.edu

Bradley R. Bebee
Director, System Software
SAIC
Advanced Systems and Concepts
3811 N. Fairfax Dr., Suite 850
Arlington VA 22203
USA
bebeeb@saic.com

Keywords: REST; HTTP; XML; XPointer; resource; subresource; addressing; direct manipulation; RDF; XTM; topic maps; semantic web; Server-Side XPointer

1 Abstract

This paper discusses one approach to a scalable, extensible logical [1] addressing scheme for information

resources and information subresources. This issue is of interest to markup authors since it provides a bridge between documents and transport. The REST [Representation State Transfer] architectural style ([REST], [FieldingWebArch]) generally provides a near identity transform for creating, reading, updating, and destroying online information resources [2]. In this model, the client may provide XML ([XML 1.0], [XML 1.1]) processing services, and optionally update the information resource to reflect changes resulting from that processing. As such, it is ideally suited to the concerns of markup authors for creating distributed hypermedia processing applications. This facility is especially valuable when the resource being addressed is very large, e.g., a busy RSS channel, a tuple space, or a triple store containing millions of triples.

For example, consider the IMDB [Internet Movie Database][IMDB], which is often taken as an example parallel to the management and analysis of information by the IC [Intelligence Community]. If the IMDB collection is treated as a resource, it contains a large typed-link structure relating a variety of different types of entities (movies, actors, directors, etc.). Researchers have performed analyses of the IMDB in an attempt use Social Network theory to predict the success of movies and the likelihood of collaboration between directors and actors [Random Graphs of Social Networks]. To support such analysis, agents require access to select subresources, that is, the analytical services for processing the links between actors, movies, and directors require the ability to identify and retrieve select subsets of the total information resource. Further, the important structural relationships to be addressed are not known a priori, but are themselves identified as evolving requirements are placed on those analytical services by their users.

In the corresponding Intelligence Community problem life is a bit more complicated: multiple organizations maintain their own "IMDBs", each using its own schema for organizing the information; multiple agents inside each organization need to be able to update their "IMDB"; the pedigree of the information is critical; and agents in each organization need to be able to perform analysis on a fused view of the total information base - as constrained by legal and procedural boundaries on what may be shared with whom.

In the course of this paper, we introduce XPointer [XML Pointer Language][XPTR] and REST and develop how they can be applied together to meet some of the requirements of the Intelligence Community using our "IMDB" as our starting point. This is a work of fiction - the real IMDB uses a completely different architecture to manage additions, validation, updates, etc. However, the organic growth, scale, and complexity of the data and the need to validate and fuse inputs from multiple agents are all part of the problem faced by the IMDB.

2 Introduction

This document explores the use of the XPointer Framework [XPTR] in combination with the REST Architectural Style [REST] and the HTTP [Hypertext Transfer Protocol]/1.1 protocol [RFC 2616]. The goal of this paper is to show that you can achieve scalable and extensible direct manipulation of very large resources using HTTP/1.1 and the XPointer Framework using a REST-ful approach. Further, since XPointer is an extensible framework, the client is able to use a logical addressing scheme [1] that is ideally suitable for identifying subresources in very different kinds of XML [Extensible Markup Language] resources (e.g., SVG [Scalable Vector Graphics][SVG 1.1], RSS [Real Simple Syndication] [RSS2.0] , Atom [Atom], RDF [Resource Description Framework][RDF/XML], and XTM [XML Topic Maps][XTM 1.0]).

The key features of the HTTP/1.1 specification that are used are the **Range** request header, the **Content-Type** entity header, and the **Accept-Ranges** response header. A client that desires to use these features

must accept a minor additional burden in how they prepare the HTTP request (the URI fragment identifier is converted into a HTTP "Range: xpointer=...." request header) and in how they process the HTTP response (the client must recognize the 206 (Partial Content) status code and, for the general case, must be able to handle the "multipart/mixed" Internet MIME [Multipurpose Internet Mail Extensions] type **[RFC 822]**, **[MIME I]**, **[MIME II]**.

Nothing in this recommendation changes the basic contract for a URI [Uniform Resource Identifier] fragment identifier, which is specified by **[URI (draft)]** and summarized below. However, the proposed approach does effectively make it possible for the client to *delegate* the evaluation of the fragment identifier to the server. To do this, the client uses the HTTP "Range" request header to convey the fragment identifier to the server. By this simple action, the client is essentially declaring that it only needs those subresource(s) that are actually addressed by the fragment indicator. A compliant server will then return only the addressed subresource(s). We call this approach, "Server-Side XPointer."

The use of XPointer expressions in URI fragment identifiers together with the HTTP GET, POST, PUT and DELETE methods is also explored. POST can be interpreted as linking a child resource from the addressed subresource or as an append or insert operation on the addressed subresource. The PUT method provides a mechanism to update the addressed subresource, but has questionable semantics when multiple subresources are addressed (at least in the general case). The DELETE method provides a natural facility for destroying multiple subresources within a single transaction (SELECT + DELETE).

In combination with the existing ETag [Entity Tag] (§ 14.19) and If-XXX mechanisms (§ 14.24 - 14.28) of the HTTP protocol **[RFC 2616]**, the client can create conditional requests, e.g., "send me the addressed data IFF it has changed since the last time I looked", or "make these updates IFF the resource state has not been modified since I read the data." The latter is useful for constructing a degree of transactional isolation across multiple requests.

Finally, it is suggested that the use of XPointer and the extensible range mechanisms of HTTP/1.1 may provide viable solution for people exploring APIs [Application Programming Interface] for semantic web services, such as the RDF Net API **[RDFNet API]** and SNAPI [Semantic Net API]**[SNAPI]**. In this context, POST naturally takes on the semantics of appending assertions (INSERT) and PUT can be developed as a transactional update mechanism, using DELETE + INSERT semantics, which makes good sense for graph models, such as a triple store or a topic map graph.

3 Background

The relationship between an Internet MIME Type, a URI, and the role of the client is spelled out in the W3C Architecture of the World Wide Web **[WebArch]**: Per [**[URI (draft)]**]

> [I]n order to know the authoritative interpretation of a fragment identifier, one must dereference the URI containing the fragment identifier. The Internet Media Type of the retrieved representation specifies the authoritative interpretation of the fragment identifier.[8]

For example, with an (X)HTML [HyperText Markup Language] representation, the agent is normally acting on behalf of a human operator who is navigating web hypermedia resources and the semantics of the fragment identifier are interpreted by a web browser as causing the viewport of the browser window

to be scrolled such that the addressed subresource is visible. As such, the fragment identifier is used by client-side processing once the resource representation is in hand.

In fact, as we show in some examples below, the HTTP client does transmit the fragment identifier as part of the Request-URI !!! Instead, the client applies the fragment identifier to resolve the reference in the retrieved representation per the semantics of fragment indicators for the Internet Media Type negotiated [7] for the representation of the resource for that GET request.

HTTP provides an extensible mechanism for clients to interact with a "range" of the resource representation. The only range-unit that is explicitly described by HTTP/1.1 is "bytes", i.e., addressing one or more byte ranges in the resource representation. However, the "bytes" range-unit is not suitable for our requirements since it does not provide an extensible mechanism for directly addressing and manipulating logical subresource(s). Instead, we turn to the XPointer Framework.

4 XPointer Framework - extensible addressing schemes

The XPointer Framework **[XPTR]** provides an extensible mechanism for addressing XML subresources. The XPointer Working Group defined a core set of XPointer addressing schemes, **[xmlns()]**, **[element()]**, and **[xpointer()]**, in addition to the basic XPointer Framework. Further, people are free to define new XPointer schemes and to adopt existing XPointer schemes for addressing subresources for specific XML grammars, e.g., SVG **[SVG 1.1]**, which defines the **[svgView()]** scheme for interpreting fragment identifiers as logical views into a rendered SVG document.

Unfortunately, of the existing XPointer schemes, the **element()** scheme is too weak and the **xpointer()** scheme is so powerful that it is rarely implemented. However, it is straight forward to declare and implement new schemes, so people should not be discouraged from adopting this approach. Further, such schemes will often be better tailored to the specific nature of the resource. Consider - it is simple to imagine an XPointer addressing scheme named **xpath()** (none exists) that would facilitate addressing subresources using the widely adopted XPath Recommendation **[XPath 1.0]**. Based on HTTP, XPath **[XPath 1.0]**, and our new **xpath()** scheme, it would be easy to develop an IMDB resource that could be maintained by interchanging XML fragments that represent metadata about entities, actors and movies.

Ideally, we want to shape our XPointer schemes with the intention that they will serve dual use. Not only will they provide a data access mechanism for subresources, but they will also make it possible for authorized agents to directly manipulate those same subresources. For example, a card catalog resource could be maintained by interchanging XML fragments that represent metadata about books in holding at a given library.

5 REST Architectural Style - direct manipulation of resource state.

REST is an "architectural style" identified and described by Roy Fielding in his doctoral thesis **[REST]** and summarized in Principled Design of the Modern Web Architecture **[FieldingWebArch]**. The term "REST" is an acronym for "Representation State Transfer" which emphasizes one of the key characteristics of that architectural style.

The following definition is excerpted from Roy Fielding's doctoral thesis. If you want to understand how Fielding uses the term "architectural style", then you should read Chapter 1 of his thesis which defines the other terms used in this definition and contrasts the way in which he defines an "architectural style"

with how other people have defined the term.

> An **architectural style** is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style.[9]

A key component of the REST architectural style is the "Simple Uniform Interface". Following Fielding in **[WAKA]**, there are five primary interface constraints that make up the "REST Uniform Interface". These are interface constraints may be related in a straight-forward manner to the concerns of markup authors:

> 1. Resource is the unit of identification [document centric with global identifiers].
> 2. Resource state is manipulated through the exchange of representations [document interchange].
> 3. [G]eneric interaction semantics [create, update, read & delete documents].
> 4. Self-descriptive messaging [supporting processing by intermediaries, e.g., caches and security firewalls].
> 5. Hypermedia is the engine of application state [hyperlink traversal plus form-based data submission].
>
> **10**

The REST architectural style evolved during work on HTTP **[RFC 2616]** - the HyperText Transfer Protocol. In many ways Fielding's thesis **[REST]** provides an idealized view of the architectural goals of HTTP. However, while REST explores HTTP, it is at the same time more abstract. REST describes an architectural style that is well-suited to very large scale distributed hypermedia applications. HTTP is a specific protocol that adheres to that architectural style in some ways and violates it in others **[WAKA]**.

The pieces of our REST architecture are: HTTP, XML, and XPointer. HTTP is a protocol that may be used to build very large scale distributed hypermedia applications. XML is a document model that may be used to interchange self-describing representations of resource state. XPointer a standard framework for expressing and evaluating subresource linking relationships in URIs. Using HTTP + XML + Linking is often equated with REST, but it is only one instance of the REST architectural style. However, it **is** what is being explored in this paper .

We use HTTP all the time with our web browsers and, as such, we only see two aspects of the protocol: (1) the use of GET to recover the representation of the state of a resource; and (2) the use of POST to send data to a web application. Together, these uses of HTTP for hypermedia navigation are sometimes called "hypermedia as the engine of application state" **[REST]**.

Recently, there has been a growing community **[rest-discuss]**, **[REST Wiki]**, **[1060.org]**, **[well-formed-web]**, **[Atom]** that is exploring other features of the HTTP protocol that provide not only for hypermedia

navigation, but also for the direct manipulation of the resource state through the interchange of representations. The basic pattern for directly managing the life cycle and state of resources through document interchange is:

POST	Create a child resource.
PUT	Update the state of the resource.
GET	Request a representation of the resource state.
DELETE	Destroy the resource.

This use is both consistent with the design and intention of the HTTP specification and corresponds closely with the world-view of "markup people".

5.1 REST Examples

5.1.1 Anatomy of an HTTP request - GET foo

Let's work through a couple of examples that illustrate how HTTP works. Let's say that you click on http://www.myorg.org/mydoc

The sample request below assumes that the client is issuing an HTTP/1.1 request, which is common nowadays, for an XML representation ("Accept: text/xml") of the resource identified by that URI. The resource is located on the Internet host "www.myorg.org" and is found at the absolute pathname "/mydoc" on that host. The protocol scheme ("http") is not represented in the request. Instead, the protocol scheme was used to select the HTTP protocol. The (implied) port number (port 80) is used when the client initiates the HTTP connection with that socket on the identified host and also appears in the Host request header (when a non-default port is specified or the port is explicitly specified).

So, the request is:

```
GET /mydoc HTTP/1.1
Host: www.myorg.org
Accept: text/xml
```

And the response is:

```
HTTP/1.1 200 Ok
Accept-Ranges: xpointer
Content-Type: text/xml

<foo/>
```

The 200 is the HTTP status code for the response [3]. Any 2xx status code indicates success. The 200 status code indicates success, but also informs the client that there is a response entity [4]. In this case the response entity is an XML document - the one that we requested. The "xpointer" value for the "Accept-Ranges" response header indicates that this resource supports the Server-Side XPointer proposal for subresource addressing - more about that later.

5.1.2 PUT foo (updating a resource)

In this example we are going to work through the use of the HTTP PUT method to update the state of a resource. The semantics for PUT here are such that the state of the resource is updated to reflect the state of the interchanged document.

Since we are sending a request entity (our XML document) this time, we need two additional pieces of information: (1) the "Content-Type" header, which tells the server the Internet MIME Type of the document being interchanged [5]; and (2) the actual document content [6].

The request:

```
PUT /mydoc HTTP/1.1
Host: www.myorg.org
Content-Type: text/xml

<!DOCTYPE foo [
  <!ELEMENT foo (bar*)>
  <!ELEMENT bar (#PCDATA)>
  <!ATTLIST bar id ID #IMPLIED>
]>
<foo>
  <bar id="a12">Hello</bar>
  <bar id="a13">World</bar>
</foo>
```

The response:

```
HTTP/1.1 204 No Content
```

The 2xx status code indicates a successful request, i.e., the state of the resource was updated and now corresponds, more or less, to the request entity. A 204 status code is used to indicate that there is no response entity - that is, the resource did not send back a document along with the response. This is typical for PUT, and for DELETE as well.

The net effect of this request is that we have changed the state of the resource so that it now has two **<bar/>** elements, each of which has an "id" attribute. The request entity also includes a **DOCTYPE** declaration which provides a means for an XML processor to understand that the "id" attribute has ID type semantics. Declaring ID type semantics is critical for many subresource addressing schemes **[xml:id]**, **[xmlIDsemantics-32]**. This example illustrates one way in which this information can be declared.

6 Addressing and manipulating XML subresources.

So, how can we use the XPointer Framework to directly address and manipulate XML subresources? The key pieces are:

Server indicates support for the XPointer Framework:

```
Accept-Range: xpointer
```

Client sends subresource request:

Range: **xpointer** = **pointer-parts**

Server provides partial content response:

Content-Type: **multipart/mixed**

6.1 Putting it all together - addressing an XML subresource with Server-Side XPointer.

The **key** proposal being made here is that the client takes the fragment identifier (the part of the URL beginning with the "#" symbol) and creates an HTTP "Range" request header using that fragment identifier.

For example, let's use the URI http://www.myorg.org/mydoc#element(a12). The only *additional* thing that the client needs to do when preparing this HTTP request is to create a "Range" header from that fragment identifier.

The "xpointer" in the Range header specifies the *range-unit*. The range-unit, **xpointer**, is what tells the server how to interpret the *range-specifier*, which in this case is **element(a12)**.

The request:

```
GET /mydoc HTTP/1.1
Host: www.myorg.org
Accept: text/xml
Range: xpointer=element(a12)
```

The response:

```
HTTP/1.1 206 Partial Content
Content-Type: multipart/mixed; boundary="simple boundary"

--simple boundary

Content-type: text/xml; charset=ISO-8859-1

<bar id="a12">Hello</bar>

--simple boundary--
```

In this case, when the server applies an XPointer processor to evaluate the XPointer expression (the **element(a12)** *range-specifier* from the "Range" header) against the negotiated representation [7] of the current state of the resource, the result is a *node set* consisting of the single XML element whose ID type attribute has the value "a12". The response therefore contains a single XML fragment, which is the serialization of that XML element. However, since there could have been multiple matched XML subresources (in the general case), the individual XML subresources are returned as the body parts of a

MIME *multipart/mixed* response entity.

The status code is still a 2xx series, indicating success, but the 206 status code is used to indicate that partial content is being returned by the server in response to the "Range" header included by the client.

6.2 Addressing multiple subresources.

For this example, we are going to make up an XPointer scheme named **xpath()**. Since this is not an official W3C [World Wide Web Consortium] XPointer scheme, it MUST be placed into an explicit namespace (we have chosen http://www.myorg.org/xpointer-schemes/xpath to illustrate the point) using the XPointer **xmlns()** scheme. This provision for explicit namespaces on XPointer schemes is part of what makes XPointer so extensible.

So, the request:

```
GET /mydoc HTTP/1.1
Host: www.myorg.org
Accept: text/xml
Range: xpointer=xmlns(xp=http://www.myorg.org/xpointer-schemes/xpath)xpath(//bar)
```

The response:

```
HTTP/1.1 206 Partial Content
Content-Type: multipart/mixed; boundary="simple boundary"

--simple boundary

Content-type: text/xml; charset=ISO-8859-1

<bar id="a12">Hello</bar>

--simple boundary

Content-type: text/xml; charset=ISO-8859-1

<bar id="a13">World</bar>

--simple boundary--
```

As you can see, the response entity now contains one body part for each matched node in the addressed XML resource. Further, the MIME body parts are presented to the client in document order - just as they would be identified if you applied that XPath expression directly to the negotiated text/xml representation of that resource. The result of applying Server-Side XPointer SHOULD be identical to the result that would have been obtained if the client applied a local XPointer Processor to the negotiation representation.

6.3 Updating subresources

Now that we can address subresources, we can also update them using the HTTP PUT method. This example simply combines the use of the Range header with the use of PUT to update the state of a resource which we already discussed above. The semantics are essentially "SELECT+UPDATE." Further, since this is expressed as a single request by the client, it is easy for the service to guarantee

transactional isolation for the request.

The request:

```
PUT /mydoc HTTP/1.1
Host: www.myorg.org
Range: xpointer=element(a12)
Content-Type: text/xml; charset=ISO-8859-1

<bar id="a12">Goodbye</bar>
```

The response:

```
HTTP/1.1 204 No Content
```

The 2xx status code indicates that the request was successful. In this case, the addressed XML element would have been replaced by the request entity (an XML fragment). If we GET the whole resource, its current XML representation is:

```
<!DOCTYPE foo [
  <!ELEMENT foo (bar*)>
  <!ELEMENT bar (#PCDATA)>
  <!ATTLIST bar id ID #IMPLIED>
]>
<foo>
  <bar id="a12">Goodbye</bar>
  <bar id="a13">World</bar>
</foo>
```

7 Scalable addressing of very large (semantic) resources.

At this point we have laid the ground work for XPointer and REST and demonstrated how these may be combined using the HTTP protocol to realize read-write subresource addressing for XML resources. Now, we would like to turn to a more specialized domain - semantic web servers.

Ideally, a "semantic web server" is a resource that is able to speak and accept any of a number of semantic web markup languages, and which can be queried using any of a number of query languages. Ideally we would like the protocol to support semantic web servers containing billions of assertions without significant application degradation.

In fact, the semantic web server is our generalization of the Intelligence Community's **[IMDB]**. It provides an extensible set of relationships (is-director-of, stars-in, played-by, reported-by, first-appearing-in) that describe and organize a universe of entities (actors, movies, directors, producers, dates, etc.) Further, each IMDB may impose a *different* organizational scheme on both the relationship types and the entities, and may use a *different* identifier for each relationship and entity. Finally, the pragmatic meaning of the identified relationships and entities may differ in subtle collection-specific ways, leading to issues in how an agent chooses to fuse these collections for a specific purpose.

7.1 Choice of Content-Type

Semantic models are graph models. Therefore, while it is possible to interchange a semantic model as an

XML document, the underlying data are best described as a set of assertions encoded as a graph **[TMGraphInXML]**, **[RDF Semantics]**, **[RDF Primer]**. Further, the interchange syntax often provides multiple means of serializing the same information. As a consequence, while we *could* query a semantic web server using addressing expressions based on an (XML) interchange syntax, we are probably going to be better off if we express those queries in terms of the underlying data models instead -- that is, if our XPointer scheme provides a *logical* addressing mechanism vs. a syntactic one.

However, there is still a role for content negotiation here. The underlying data models for RDF and Topic Maps are not the same, at least at the level which most query languages consider. Therefore an agent would probably choose a different logical addressing scheme depending on the negotiated Internet MIME type for the representation. Even in RDF based languages, there is enough variation both in the semantics and the ``logical syntax" (or data model) that even if a query language works for both, say, RDF and OWL [Web Ontology Language]**[OWL]**, the agent would want to decide whether to query against the store as RDF, or as OWL, or one of the species of OWL!

Since the semantic markup languages are oriented toward the interchange of graph structures, and many semantic stores represent large aggregation of documents, or of information that can be usefully organized as documents, it would be nice to return only subgraphs. If we address only subgraphs, we can probably get away with a simple (non-multipart) MIME type, i.e., "application/rdf+xml" when using a logical addressing scheme for RDF resources. (Based on **[draft-xtm-mime-01]**, the MIME type would be "application/xtm+xml" for topic map resources.) This dramatically reduces the implementation burden since neither the client nor the service needs to support the "multipart/mixed" MIME type.

This is not perfectly generalizable, especially with more expressive languages like OWL (and more expressive query languages). Because OWL can represent incomplete information, we might be able to answer a query without being able to provide a subgraph which is the answer to that query. For example, suppose your resource defined Person as the transitive closure of parentOf and asserted that Mathonwy was the ancesterOf Bronwyn. If the resource were queried (?X parentOf ?Y), it is not obvious what "sub"-graph to return, even if we could decide on a suitable graph that wasn't just the encoding of variable bindings. Much depends on the level of your logical view -- addressing the structure using XPath of a PSVI [Post-Schema Validation InfoSet] of a RDF/XML document can be considered logical addressing when compared to byte ranges, while querying the graph structure of that document (sans most semantics) is itself an abstraction over those infosets.

Furthermore, existing RDF query languages (e.g., RDQL **[RDQL]** and variants, path languages like Versa or GraphPath, etc.) tend to return variable bindings, where the bound values correspond to nodes of the graph. In these cases, while the fallback would be "multipart/mixed" MIME type, one might also be satisfied with a graph based "reified" representation of the results. Such reification is not uncommon in Semantic web circles, and the desire for such is even more common. Or, the node selection might trigger the return of other metadata about the selected nodes, a la, the URIQA [URI Query Agent] **[URIQA]** proposal. In that case, the query language would be used to select a set of "root" nodes while some other mechanism would be used to determine which assertions about those nodes would be returned. In the case where the query pattern determines a subgraph, that might be the natural subgraph to return. These considerations suggest that *addressing* has different requirements and desiderata than general query.

In order to keep things simpler, we are going to explore subresource addressing for RDF and XTM separately. However, we are carrying an underlying assumption that the same resource (same URI) is exposing either RDF/XML or XTM to the client based on HTTP content negotiation. The way this works is that the client indicates the Internet MIME type of the representation using the HTTP "Accept" header. We have been doing this all along in our examples, but if you notice below, you will see that the

same resource is responding with either RDF or XML depending on the value of the client's "Accept" header. For more information on harmonization of the RDF and XML Topic Map data models, see **[TMRDF]**, **[RDF&TM]**, and **[Integration of TM&RDF]**. Presumably any update using one representation would be reflected in the other -- that is, the resource has some internal state, but it decides how to expose that state to the client based on content negotiation.

7.2 RDF Query Language

At this time, there is a large and growing body of query languages for RDF and XTM. However, since XPointer is an extensible framework, we can pick any query language we like and develop a notation for that query language that is consistent with the XPointer Framework. With that out of the way, we will be able to encode the query in a URI and use the techniques described above to address subresources in our semantic store.

For our examples, we picked the SeRQL [Sesame RDF Query Language]**[SeRQL]** query language. SeRQL is in the RDQL family of RDF query languages, and thus is loosely modeled on SQL, i.e., RDQL queries follow the SELECT...FROM...WHERE pattern. RDQL was developed by Andy Seaborne at HP Labs and is, perhaps, the most popular and prominent of the many RDF query languages, with its specification recently accepted as a W3C member submission. SeRQL has a number of convenient extensions including the "CONSTRUCT" clause. CONSTRUCT is like SELECT except that instead of a list of bindings, CONSTRUCT returns a (constructed) graph. For example:

```
CONSTRUCT {Evidence} <foo:supports> {Hypothesis}
FROM {Hypothesis} <foo:hasSupport> {Evidence}
USING NAMESPACE foo = <!http://www.example.org/foo#evidentiary-model>
```

Note that the returned graph for the above query could easily not be a proper subgraph of a store. The store, for example, might only contain foo:hasParent claims, with no rules or ontological definitions to generate the inverse. Of course, merely by supporting CONSTRUCT, the store is letting itself be extended willy nilly by clients. Or rather, by client URI designers.

CONSTRUCT also supports * so that, if possible, the CONSTRUCT will return a graph created by instantiating the query itself. If lucky, that will be an actual subgraph of the store. For example:

```
CONSTRUCT *
FROM {SUB} <rdf:type> <rdfs:Class> ;
     {SUPER} <rdf:type> <rdfs:Class>
     {SUB} <rdfs:subClassOf> {SUPER}
```

(Note that "rdf" and "rdfs" are assigned to the obvious namespaces by default.)

7.3 RDF Examples.

7.3.1 RDF Query Language and XPointer Scheme

For these examples, we define a custom XPointer scheme, called "graph-serql", based on the CONSTRUCTive subset of SeRQL. We restrict our scheme to CONSTRUCT based queries so that the resource addressed is always a graph. We further restrict our scheme to CONSTRUCT * where the store can always return a subgraph identified by the query (hence, minimal expressivity and inference). In a real system with a more liberal query language and expressive backend, there would be many opportunities for interesting sorts of negotiation. For example, it could be useful to keep distinct the return MIME types so that if the server determined that the query could not return a graph, but the only acceptable type was "application/rdf+xml", it would fault.

7.3.2 Query

Since this is not a W3C specified XPointer scheme, the **rdf-query()** scheme will need to be in an explicit namespace, we use http://www.myorg.org/xpointer-scheme/rdf-query#. Therefore we use the **xmlns()** scheme to setup the namespace context so that our **graph-serql()** scheme is correctly recognized by the XPointer processor. Once the namespace context has been established, we just wrap up the query and set the whole XPointer expression as the value of the "Range" header. When the request is received by the server it will use an XPointer processor to evaluate that express on behalf of the client and ship back the results.

The request:

```
GET /myStore HTTP/1.1
Host: www.myorg.org
Accept: application/rdf+xml
Range: xmlns(q=http://www.myorg.org/xpointer-scheme/rdf-query#)
       q:graph-serql(
          CONSTRUCT *
          WHERE {x} foaf:name "John Smith";
                {x} foaf:mbox {mbox}
          USING NAMESPACE foaf = <!http://xmlns.com/foaf/0.1/>
       )
```

The response:

```
HTTP/1.1 206 Partial Content
Content-Type: application/rdf+xml

<rdf:RDF xmlns:rdf="...">...</rdf:RDF>
```

What is interesting about this example is that multiple subresources are being returned using a simple (vs. multipart) MIME type. We can get away with this since (a) we are using logical (vs. syntactic) addressing scheme; (b) we are presuming that any subresources of an RDF graph are also an RDF graph (i.e., you can't address below the level of a triple); (c) any RDF graph can be interchanged as a single "application/rdf+xml" document.

7.3.3 Update

If the semantic web server performs no inference at all, then update of a store is essentially equivalent to editing an RDF/XML document and is quite straightfoward. Again, it will typically be much easier to update the graph model than the model of the XML serialization. However, if the server performs significant inference, then update becomes much trickier. For example, adding an assertion might cause the knowledge base to become inconsistent and thus unreliable (assuming the store's formalism is rich enough to express contradictions). We can mitigate this by checking for consistency after each assertion (or batch of assertions), but this is a very computationally expensive operation. Deletes are much worse. If the assertion we are trying to delete is implied by the rest of the knowledge base, effective deletion might require deleting a further set of assertions (those which entail our target). There may be many different candidate sets of support for our assertion, leading to a non-deterministic choice of what else to delete. The most popular existing protocol for remote access for stores supporting languages with the expressivity of OWL, the DIG [DL Implementors Group] interface **[DIG][DIG Interface]**, only allows inserts to the store. Effectively supporting deletes is a hard open problem.

7.4 Topic Map Query Language

In querying Topic Maps we adopt the same conceptual approach to that of addressing RDF models. We could utilize several maturing topic map query languages, AsTMa? **[AsTMa?]** and Tolog **[Tolog]** to illustrate semantic addressing within this framework. We expect the result of querying the topic map to be an XTM instance that is a representation of the topic map sub graph retrieved from the entire topic map. The TMQL [Topic Map Query Language]**[TMQL]** standardization is further investigating alternative representations as valid responses to Topic Map queries.

We consider Topic Maps to consist of 3 atomically retrievable components, although there are many more addressable ones. The Topic Map, a Topic and an Association are the basic components in the Topic Map model. Any responses that are XTM instances must be a valid XML Topic Map and consist of these three components. Unlike RDF, Topic Maps does not collapse identity on Topics. Thus it is necessary to use a Topic Map fragment algorithm when creating an XTM representation of a partial topic map.

7.5 Topic Map Examples

The examples in this section show how the framework we have introduced can be used to address and retrieve subresource of a semantic resource. We consider the entire topic map to be one resource and the topics and associations within it to be subresources. We first illustrate a basic query to compare with the RDF examples above. We then discuss the possible update semantics of this approach and relate them to abstract semantic update models such as SNAPI **[SNAPI]** and RDF Net API **[RDFNet API]**.

7.5.1 Query

Since this is not a W3C specified XPointer scheme for Topic Maps, the **tm-query()** scheme will need to be in an explicit namespace, we use http://www.myorg.org/xpointer-scheme/tm-query. Therefore we use the **xmlns()** scheme to setup the namespace context so that our **tm-query()** scheme is correctly recognized by the XPointer processor. Once the namespace context has been established, we just wrap up the query and set the whole XPointer expression as the value of the "Range" header. When the request is received by the server it will use an XPointer processor to evaluate that expression on behalf of the client and ship back the results.

The request:

```
GET /myStore HTTP/1.1
Host: www.myorg.org
Accept: application/rdf+xml
Range: xmlns(q=http://www.myorg.org/xpointer-scheme/tm-query)
       q:tm-query( WHERE
                   exists [ $toc @ S : * ]
                   RETURN
                   )
```

The response:

```
HTTP/1.1 206 Partial Content
Content-Type: application/xtm+xml

<xtm:topicMap xmlns:xtm="...">...</xtm:topicMap>
```

In this example we are retrieving a subgraph of the topic map. In conjunction with an appropriate fragment algorithm, not given here, we are able to retrieve a valid topic map graph as an XTM instance.

7.5.2 SNAPI (Semantic Network API) Binding

The RDF Net API, and more generally SNAPI, provides a basic set of operations based on REST principles that provide for the update, delete and construction of an RDF model or Topic Map. The protocol is defined in an abstract fashion that does not prescribe any specific binding. The approach presented here provides one alternative way of binding into the abstract protocol.

The SNAPI protocol is a genericised version of the RDF Net API but the basic principles are the same. There is a small collection of operations, Query, GetKnowledgeStructure, InsertKnowledgeStructure, RemoveKnowledgeStructure, PutKnowlegeStructure and UpdateKnowledgeStructure. In the case of Topic Maps the knowledge structure in question is a Topic Map fragment and in RDF it is an RDF Model. The operations all work on a by value basis. This means that in delete for example the values to delete are passed up by value. In addition, this protocol has no modification semantic whereby updates are done in the form of delete followed by an add. The following sections discuss the core protocol and relate it to how it can be implemented in terms of this framework. We also discuss how it can extend and refine the semantics of the protocol.

The basic Query operation in SNAPI is directly analogous to the example provided above. In this case the new XPointer scheme is used to wrap the query for the given topic or topics. SNAPI does not express the scope of the result, i.e. how many topics should be returned. Thus any implementation that returns a valid XTM fragment is compliant.

The SNAPI "Update" operation is perhaps the most interesting operation with regard to this framework. Update replaces the knowledge identified with information provided in the body of the request. The identification of the subresources to be replaced matches the intended use of the extended XPointer scheme in this framework. What this does is provides greater clarity of the protocol. In other bindings the query to identify the subresource has been done in a non-standard fashion. This framework provides an semantic basis on which to base a binding to the abstract protocol. It should be noted that update is defined in terms of a DELETE and POST. However, it could be defined in terms of a PUT where the resources removed are subresources of the Topic Map.

The DeleteKnowledge operation can make use of the machinery provided to identify and delete subresources of the Topic Map, i.e. Topics and Associations. Again, this provides a clearer understanding and communication of the semantics of the operation when it has a concrete binding such as this.

8 State of this research, critical commentary, and related work.

8.1 How can I know when a resource supports Server-Side XPointer?

The HTTP "Accept-Ranges" response header will include the "xpointer" range-unit token. You can test this using the HTTP HEAD method. The HEAD method is just like GET, but only the metadata about the resource is returned. In our case, this means that we will be able to find out whether or not the resource declares support for the "xpointer" range-unit.

The request:

```
HEAD /mydoc HTTP/1.1
Host: www.myorg.org
Accept: text/xml
```

The response:

```
204 No Content
Accept-Ranges: xpointer
```

The presence of the "Accept-Ranges" header with an "xpointer" range-unit provides the indication that the resource supports Server-Side XPointer. Also, while the examples above have not shown it, the "Accept-Ranges" header SHOULD be included in every response from a resource that supports this proposal.

8.2 Can I bookmark my queries?

Yes, and no. Bookmarks are representations of the URL and include the fragment identifier which is used to encode the XPointer expression - as such they are normal URLs and can be bookmarked. However, evaluating the XPointer expression in an efficient manner requires that the client bind the expression onto the HTTP "Range" header in order to address the indicated logical subresource(s). Failure to do this on the part of the client means that the client is requesting a representation of the entire addressed resource.

In practice, when deploying such very large resources, people may want to deliver metadata about the resource as the default resource representation. This metadata could be similar to that currently provided for online databases and might include a link to a hypermedia application that provided some means for navigating views of the resource state. E.g., a web application that makes it easy for users to ask certain kinds of questions of the semantic store.

8.3 Unresolved issues

8.3.1 The Content-Range response header.

While the use of the Content-Range response header is indicated by the HTTP/1.1 specification, there is certainly room to clear up what its semantics are when using multiple pointer parts, e.g., "#element(a12) xpointer(/foo/bar)", or when using a logical addressing scheme (vs. a syntactic one). For example, the server response could indicate which pointer part matched or it could provide an xpointer (or other range-unit expression) which it choose to consider as canonical.

8.3.2 Should we register a node-list MIME type?

This proposal suggests the use of the MIME "multipart/mixed" type for the interchange of ordered XML node sets, such as those produced by an XPath processor. It might be convenient to register a MIME type that specialized "multipart/mixed" to indicate explicitly that the parts were XML information items, e.g., "multipart/xml-node-set".

8.3.3 Content negotiation.

While it is not drawn out specifically in the text above, this proposal takes the position that content

negotiation occurs before the server applies the XPointer Framework, and, as a consequence, that the negotiated content type is found on the individual body parts in the multipart/mixed response entity. Some people may consider this "broken", in which case the client would need to negotiated directly for "multipart/mixed", but that leaves the client without a means to negotiated the representation to which the XPointer expression is applied by the server and results in an unworkable approach for resources that what to support content negotiation.

For example, consider a weblog that supports content negotiation. Depending on whether the client prefers "application/rss+xml" or "application/atom+xml", they can negotiate either an RSS or an Atom view of the weblog. Further, if the client wants to use subresource addressing as outline above, it is critical that they first negotiate the Internet MIME type of the representation since: (a) the semantics of the fragment identifier are determined by the Internet MIME type of the representation; and (b) the syntax of the two representations will be different, so different addressing expressions will be required depending on which view is negotiated.

8.4 Impact on HTTP proxies

HTTP proxies are responsible for a variety of architectural features, including caching and security for HTTP resources. Multiple implementations and testing would be required to judge the impact that this recommendation would have on HTTP proxies. There are several constraints in the HTTP/1.1 specification that are designed to guide the implementers of HTTP proxies that address: (a) multipart/mime; and (b) the use of the HTTP Range header.

In **[Re: HTTP Methods]**, Roy Fielding cautions

> Range is very hard to do right, and almost always breaks across proxies. Generally speaking, it should only be used for byte ranges, and only then for the sake of finishing incomplete downloads or working around limited client buffers.[11]

8.4.1 Caching

Caching facilitates a scalable content distribution network by decentralizing access to leased representations of resource state. The HTTP protocol provides a variety of mechanisms that a server may use to describe this lease, including that NO lease is provided, i.e., the representation is not cacheable. At the same time, HTTP provides the client with mechanisms that it can use to drill through caching proxies and obtain an authoritative representation of the state of a resource.

Much of the protocol complexity concerning the HTTP **Range** has to do with breaking transparency in caching proxies in order to achieve a more efficient (scalable) content distribution network. In particular, example, for a 206 (Partial Content) response when the range-unit is "bytes" and the Content-Type is "multipart/byteranges", some HTTP/1.1 caches break transparency and, where permitted by the "lease terms", combine cached representations using adjacent or overlapping byte ranges.

However, for the purposes of this recommendation, we would recommend that HTTP proxies NOT break transparency and NOT attempt to combine physically or logically adjacent representations within cache.

Instead, if a resource wants to provide a cacheable representation, we recommend that it redirect the client to a URI whose representation provides a cacheable response to the subresource query, for example by responding with a 303 (See Other) status code and including the URI of a secondary resource that is responsive to the client's query. See section 10.3.4 of RFC 2616 **[RFC 2616]**.

The HTTP/1.1 protocol (§ 13.10) makes provisions to support the automatic invalidation of cached resource representations after updates or deletion operations using POST, PUT or DELETE. This automatic invalidation facilitates the use of these methods for the direct manipulation of resource state. For example, after using PUT to update the state of a resource, any caching proxy which observes that request will invalidate any cached representations for that resource. This has the desired effect that a subsequent GET will obtain a current representation of the resource, which will then be cached (if allowed) for other clients. However, there are limitations. For example, a cache that does not see an update or deletion request will not invalidate its representation.

The question of correct (or at least good) cache control behaviors for the direct manipulation of subresources is closely related to the question of cache control for a resource that exposes different representations using different URIs. In both cases, a caching proxy needs to be able to determine the (parameterized) dependency relationships between the resource update and the cached representations that should be invalidated as a result of that update or delete operation.

This is a much larger issue facing semantic web servers, especially those that hope to expose both RDF and XTM facing service aspects. Unless the service reveals the dependency among the subresources in the response, a caching proxy will be unable to detect when an update on one subresource (or in one view) causes another subresource to become stale. If the server does not expose the dependency relationships, then it must either accept that stale representations will be provided to clients at least some of the time or it must disable caching altogether. This is very much a quality of service issue in which the role of the protocol is to enable the service and the client to achieve the compromise that best reflects their requirements.

8.4.2 Security

HTTP provides an extensible architecture for resource level security, e.g., through the configuration of HTTP proxies and network firewalls. In an HTTP security scenario, a proxy will typically examine three things: (1) the principle (if authenticated); (2) the URI of the addressed resource; and (3) the requested method (GET, POST, PUT, DELETE, etc.). Based on those three items, and perhaps on whether or not the channel is secure, the security agent will make a decision on whether or not to reject the client's request, to redirect the client, or to challenge the client to authenticate themselves.

This approach facilitates security policies that control which agents may read or directly manipulate the state of which resources. For example, a workflow agent may be issued credentials such that it can update the state of various business resources while other users may have read-only access to the same resources and agents not belonging to the host organization may be denied any access to the resource.

It is problematic to apply security constraints at the network level at a finer grain than the resource since there are multiple ways to ask for a given subresource, which could result in security loopholes that are difficult to identify and to plug. The same problem exists when there is more than one URI for a given resource, but it is made many, many times worse when you consider securing an extensible subresource addressing mechanism.

While it would be possible to expand the set of properties that a security proxy considers for a request,

the multiplicity of different ways in which a client would ask for the same data in an extensible subresource addressing scheme could make it difficult to properly permit or deny a given request. One approach worth exploring would be to redirect the client to a URI which provided only a view of the addressed subresource and establishing security constrained based on those secondary URI. Ideally, those secondary URIs would be generatively produced in terms of some canonical subresource identification scheme known internally to the server and coordinated with the security agent.

8.5 Loosely Coupled Design

By combining server-side XPointer and HTTP Content Negotiation (CONNEG) we are able to achieve loosely coupled designs that are highly evolvable. For example, the current IMDB is backed by a set of ASCII files that are mirrored onto various servers and also exposed through a hypermedia interface on the IMDB web site. By introducing an XML Schema for interchanging the IMDB data and creating an XPointer scheme for addressing into the IMDB, the existing implementation could be evolved to also expose itself to agents using that XML Schema. However, the IMDB implementation would not be locked into that XML Schema and it could easily be evolved into a semantic web server, e.g., based on an RDF store or a topic map graph vs. a purpose-specific RDBMS Schema. Clients could continue to use the XML Schema for interchange, but they could also interchange with the IMDB service using an RDF Schema designed to expose the relationships modeled by today's IMDB. The advantage of an RDF or XTM-based interchange is that it is inherently extensible. A client can add new *kinds* information into the IMDB simply be interchanging RDF statements whose semantics are declared by some third party RDF Schema. The choice of interchanging the XML Schema vs. RDF/XML or XTM is one of suitability for an application specific purpose vs. extensibility. Using Server-Side XPointer and CONNEG we don't have to design in that choice - the client can decide for themselves on a request by request basis.

8.6 Alternative approaches

In this section, we would like to consider some alternative ways of approaching either the general problem of subresource addressing or the specific problem of scalable data access for semantic web servers.

8.6.1 Why not use the query string?

It suprised me, but a search of the HTTP/1.1 specification (RFC 2616) shows that HTTP/1.1 says nearly nothing about the meaning of the URL query string. Given this lack of constraint, one could use the query string to pass in a logical subresource address. However, this is likely to confound pre-existing application semantics for GET using the URL Query String. Further, unlike the XPointer Framework, this approach does not provide for extensible orthogonal addressing schemes. Finally, the XPointer Framework itself is explicitly a proposal for the semantics of the URL fragment identifier. This proposal suggests that the client simply indicate the logical subresources to be addressed. Getting the XPointer expression into the URL query string would mean re-writing the URL, not just adding information to the HTTP request.

There was a day when the practical length limits of a URI was a real concern. Today most software can handle relatively large URIs (4kb or more) (see § 3.2.1 and § 10.4.15 of **[RFC 2616]**). However, it is worth noting that the proposed approach places the fragment identifier on the Range header, and it does not occur as part of the Request-URI *at all*. This means that using XPointer together with the HTTP Range header, not only do you keep from intruding on existing application semantics for the URL query string, but the length of the Request-URI does not grow with the size and complexity of the query being

made.

8.6.2 Use the HTTP Link header.

Roy Fielding offered a critique of this recommendation and a counter-proposal based on the HTTP **Link** header [Re: HTTP Methods] which would allow an agent to discover an alternative, parameterized URI by which the resource would expose access to subresources.

The HTTP **Link** header was described in RFC 2068 [RFC 2068], which was an earlier draft of HTTP/1.1, however it is not included in RFC 2616 [RFC 2616], which is the current version of the HTTP/1.1 specification. The semantics of the **Link** header are the same as those of the HTML [HTML 4.01 Specification]<LINK> element. The HTTP LINK and UNLINK methods from RFC 2068 were also dropped in RFC 2616. Those methods provided the client with an opportunity to directly manage the Link headers associated with a resource.

In [Re: HTTP Methods], Fielding writes:

> A better solution is to let the server define its own URI space such that it gives the client a license to dig deeper. E.g., a link header field like
>
> Link: <http://example.com/big/resource;xpf=>; rel="xpointable"
>
> tells the client the URI
>
> http://example.com/big/resource;xpf=xpath(//item/3)
>
> can be constructed to identify the resource that is described by that xpath. However, personally, I would avoid use of anything other than id's, since manipulation of the XML data tree will create an implementation dependency between client and server.[12]

In general, the HTTP Link header, the HTML LINK element, and the XLink Recommendation each provide a means to express inter-document links with labeled arc roles. There is an overlap with the role of RDF and XML Topic Maps to specify such labeled linking relationships. It would be a good thing if a standard was established that showed how the same linking information could be represented using each of these approaches. For example, see [Expressing Dublin Core in HTMK/XHTML] for a recommendation on how to encode the Dublin Core Metadata using the HTML <LINK> and <META> elements and [Dublin Core in RDF/XML] for a recommendation on how to encode the Dublin Core Metadata in RDF/XML.

8.6.3 Avoid data structure dependency in addressing

In a follow-up message [Re: HTTP Methods], Fielding highlights the concern with a data dependency in addressing schemes:

> Let's say that we come up with a new data format standard, TBTSSB, which is so incredibly better than XML that everyone wants to switch. If past references to those resources are via assignable names, rather than data structure, then nothing needs to break in the transition. We simply make sure that the new representation also defines those names.
>
> In contrast, if the clients are assuming that the server represents the resource as XML, then moving away from XML will cause them to break, and no amount of content negotiation can help that situation because the addressing mechanism itself has become format-specific.
>
> Note that, in spite of the fact that they are often called URLs and frequently considered "addresses", every bit of indentifying information in an http URI is an assignable name. That is, until someone decides to use a fragment identifier that is format-specific. [13]

I unpack this as being several inter-related design issues. The first is how linked the addressing scheme is to the syntax of a resource representation. For example, the Groves paradigm emphasizes property sets, which may be coupled to the syntax or may be related to more abstract characteristics - so at some level this is a design decision within the architecture. The more invariant the addressing scheme with respect to the possible representations, the greater the longevity and evolvability of the web.

The second is that the client should be able to address the same (sub-)resource using the same (fragment) identifier regardless of the representation which they negotiate with the server. However, I think format specific fragment identifiers can be quite valuable (e.g., the svgView()) - and I think that the use of redirection may provide a solution that maintains the assignable name characteristic of the architecture. So, in this approach the server could choose to transparently redirect the client to a URI that serves as an "assignable name" for the addressed subresource(s). In this manner, any fragment identifier scheme (whether syntactic, abstract, or simply an opaque name) can be converted into an assigned name chosen by the service.

However, this approach clearly raises the level of complexity of the resource realization, since it must now maintain state about the secondary resources that are exposed to clients through redirection, including what dependencies they have on the resource from which they were derived, how they are secured, and the terms of their caching contracts and other service provisioning concerns.

This does not look like an either/or choice. Good choices for subresource addressing schemes can minimize the dependency on the syntax while transparent redirection can make it possible to have uniform addressing over subresources through assigned names.

8.7 What is the relationship to Groves? [Groves]

The present proposal provides for an extensible read/write addressing scheme for logical subresources using the HTTP protocol underlying the World Wide Web. The Grove paradigm provides for an extensible property set view of information items. One way to leverage the Grove paradigm for this proposal would be to define an XPointer scheme, e.g., grove(). In the best case, this would provide for Grove addressing on the World Wide Web, and for an updatable view of information items irrespective of their data model.

9 End notes

12345678910111213

Notes

1. By a logical addressing scheme, we mean one that is based on an abstraction of the notation being interchanged. The level of abstraction can vary greatly, from addressing the XML data model to addressing an abstract property set, a topic map graph or the RDF data model.

2. Owing to both round-trip problems with XML and service-specific side effects.

3. HTTP divides up such application level status codes into five families: Information (1xx), Success (2xx), Redirection (3xx), Client Error (4xx), and Server Error (5xx). The HTTP specification includes clear prose descriptions of each status code. You should consult these codes early and often when architecting a hypermedia application using HTTP.

4. HTTP uses the term "entity" to refer to the document being interchanged. It is a "request entity" when the client sends the document to the server and a "response entity" when the server sends the document to the client.

5. HTTP assumes that the character encoding is ISO-8859-1 unless the Content-Type explicitly specifies otherwise.

6. While HTTP/1.1 does provide for streaming content when the size of the entity is not known (or is not specified), we would normally also send a "Content-Length" header, which specifies the size of the document being interchanged in octets (8-bit bytes), not characters.

7. The semantics of the fragment identifier is defined in terms of negotiated Internet MIME type and is applied to the negotiated representation the resource. When the client is smart about the use of fragment identifiers, this means that they can correctly choose the right fragment identifier for the desired content type, e.g., RSS vs. Atom. This is done using the HTTP Accept request header.

   However, if the client attempts to apply a fragment identifier is not valid for the negotiated Internet MIME type, then the evaluation of the fragment identifier will fail. For example, one person author's a link into an "SVG" resource using the svgView() fragment identifier scheme but another person is using a browser that does not support SVG so the client automatically negotiates a JPEG image instead. The svgView() scheme is not defined for JPEG, and the browser will not be able to establish the correct viewport onto the retrieved image.

8. **[WebArch]**, section 3.3.

9. **[REST]**, section 1.5.

10. **[WAKA]**, page 10.

11. **[Re: HTTP Methods]**, paragraph 2.

12. **[Re: HTTP Methods]**, paragraphs 8 - 12.

13. **[Re: HTTP Methods]**, paragraphs 4 - 6.

Bibliography

[REST] Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.
http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[FieldingWebArch] Principled Design of the Modern Web Architecture. Roy T. Fielding and Richard N. Taylor. ACM Transactions on Internet Technology, 2(2):115--150, May 2002. ACM Press, New York, NY, USA.
http://www.ics.uci.edu/%7Efielding/pubs/webarch_icse2000.pdf. This paper is pretty much an excerpt of Chapter 5 of Roy Fielding's thesis and is an excellent place to begin understanding the REST Architectural Style.

[RFC 2616] Hypertext Transfer Protocol -- HTTP/1.1, The Internet Society, 1999, Fielding, et alia. (Obsoletes RFC 2068).
http://www.ietf.org/rfc/rfc2616

[RFC 2068] Hypertext Transfer Protocol -- HTTP/1.1, The Internet Society, 1999, Fielding, et alia. (Obsoleted by RFC 2616)
http://www.ietf.org/rfc/rfc2068

[XPTR] XPointer Framework. Ed. Paul Grosso et al. 13 Nov 2002. World Wide Web Consortium. 25 Mar 2003
http://www.w3.org/TR/xptr-framework/

[xmlns()] XPointer xmlns Scheme. Ed. Steven J. DeRose et al. 13 Nov 2002. World Wide Web Consortium. 25 Mar 2003
http://www.w3.org/TR/xptr-xmlns/

[element()] XPointer Element Scheme. Ed. Paul Grosso et al. 13 Nov 2002. World Wide Web Consortium. 25 Mar 2003
http://www.w3.org/TR/xptr-element/

[xpointer()] XPointer Scheme. Ed. Steven DeRose et al. 10 Jul 2002. World Wide Web Consortium. 19 Dec 2002
http://www.w3.org/TR/xptr-xpointer/

[XTM 1.0] XML Topic Maps 1.0. Ed. Steven Pepper and Graham Moore. 2001. Topic Maps.Org. 08 Aug 2001
http://www.topicmaps.org/xtm/1.0/

[RDF/XML] RDF/XML Syntax Specification. Ed. David Beckett. 15 Dec 2003. World Wide Web Consortium. 10 Feb 2004
http://www.w3.org/TR/rdf-syntax-grammar/

[RDF Semantics] RDF Semantics. Ed. Patrick Hayes. 15 Dec 2003. World Wide Web Consortium. 10 Feb 2004
http://www.w3.org/TR/rdf-mt/

[RDF Primer] RDF Primer. Ed. Frank Manola and Eric Miller. 15 Dec 2003. World Wide Web Consortium. 10 Feb 2004
http://www.w3.org/TR/rdf-primer/

[RDFNet API] RDF Net API - W3C Member Submission 2 October 2003 by Andy Seaborne and Graham Moore
http://www.w3.org/Submission/2003/SUBM-rdf-netapi-20031002/

[RDF DAWG] RDF Data Access Working Group. Ed. Eric Prud'hommeaux and Dan Connolly. 08 Jul 2004
http://www.w3.org/2001/sw/DataAccess/

[Groves] "Groves, Grove Plans and Property Sets." Cover Pages. 2002. Oasis. 2 Jun 2004.
http://xml.coverpages.org/groves.html

[XML 1.0] Extensible Markup Language (XML) 1.0 (Third Edition). Ed. Tim Bray et al. 30 Oct 2003. World Wide Web Consortium. 04 Feb 2004
http://www.w3.org/TR/2004/REC-xml-20040204

[XML 1.1] Extensible Markup Language (XML) 1.1 (Third Edition). Ed. Tim Bray et al. 05 Nov 2003. World Wide Web Consortium. 15 Apr 2004
http://www.w3.org/TR/2004/REC-xml11-20040204/

[URI (draft)] Uniform Resource Identifier (URI): Generic Syntax. Ed. T. Berners-Lee et al. 2004. The Internet Society. May 2004
http://gbiv.com/protocols/uri/rev-2002/rfc2396bis.html

[WebArch] Architecture of the World Wide Web. Ed. Ian Jacobs. 09 Dec 2003. World Wide Web Consortium. 05 Jul 2004
http://www.w3.org/TR/2003/WD-webarch-20031209. See the discussion of fragment identifiers http://www.w3.org/TR/2003/WD-webarch-20031209/#fragid and the discussion of the relationship between media types and fragment identifiers http://www.w3.org/TR/2003/WD-webarch-20031209/#internet-media-type. Recent language in the web architecture document talks about the meaning of the fragment identifier in terms of "secondary resources", in contrast to the XPointer Framework, which discusses the use of fragment identifiers to address subresources, but does not define either "resource" or "subresource".

[SVG 1.1] Scalable Vector Graphics (SVG) 1.1 Specification. Ed. Jon Ferraiolo et al. 15 Nov 2002. World Wide Web Consortium. 14 Jan 2003
http://www.w3.org/TR/SVG11/

[svgView()] SVG Fragment Identifiers. Ed. Jon Ferraiolo et al. 15 Nov 2002. World Wide Web Consortium. 14 Jan 2003
http://www.w3.org/TR/SVG11/linking.html#SVGFragmentIdentifiers

[XPath 1.0] XML Path Language (XPath) Version 1.0. Ed. James Clark and Steve DeRose. 21 Apr 1999. World Wide Web Consortium. 16 Nov 1999.
http://www.w3.org/TR/xpath#function-id, and the XPath syntax for referencing IDs http://www.w3.org/TR/xpath#function-id.

[RFC 822] Standard for the format of ARPA Internet Text Messages. Ed. David H. Crocker. 13 Aug 1982. The Internet Engineering Task Force. 06 Jul 2004
http://www.ietf.org/rfc/rfc822

[MIME I] Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. Ed. N. Freed and N. Borenstein. November 1996
http://www.ietf.org/rfc/rfc2045

[MIME II] Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. Ed. N. Freed and N. Borenstein. November 1996
http://www.ietf.org/rfc/rfc2046

[IMDB] Internet Movie Database. 2004. Internet Movie Database, Inc. 12 July 2004
http://www.imdb.org/

[Random Graphs of Social Networks] Proc Natl Acad Sci U S A. 2002 February 19; 99 (Suppl. 1): 2566-2572
http://www.imdb.org/

[draft-xtm-mime-01] The 'application/xtm+xml' Media Type. Ed. P. Durusau et al. Feb 2004. The Internet Society. Aug 2004
http://www.topicmapping.com/xtm_mime/draft_xtm_mime.html

[HTTP Extension Framework] An HTTP Extension Framework. Ed. H Nielsen et al. Feb 2000. The Internet Society. 01 Jul 2004
ftp://ftp.isi.edu/in-notes/rfc2774.txt

[HTML 4.01 Specification] HTML 4.01 Specification. Ed. Dave Raggett. 24 Apr 1998. World Wide Web Consortium. 24 Dec 1999
http://www.w3.org/TR/REC-html40/, but also see "Document relationships: the LINK element"
http://www.w3.org/TR/REC-html40/struct/links.html#edef-LINK and "Linking to Stylesheets with HTTP headers"
http://www.w3.org/TR/REC-html40/present/styles.html#h-14.6

[Expressing Dublin Core in HTMK/XHTML] Expressing Dublin Core in HTML/XHTML Meta and Link Elements. Ed. Andy Powell. 30 Nov 2003. University of Bath. 01 Jun 2004
http://dublincore.org/documents/dcq-html/

[Re: HTTP Methods] Fielding, Roy. "Re: HTTP Methods." World Wide Web Consortium. 27 Feb 2004. Comment on the proposal to use the XPointer Framework in conjunction with the HTTP **Range** header.
http://lists.w3.org/Archives/Public/www-tag/2004Feb/0075.html

[Re: HTTP Methods] Fielding, Roy. "Re: HTTP Methods." World Wide Web Consortium. 19 Apr 2004. Clarification of comment on the proposal to use the XPointer Framework in conjunction with the HTTP **Range** header.
http://lists.w3.org/Archives/Public/www-tag/2004Apr/0019.html

[Dublin Core in RDF/XML] Expressing Simple Dublin Core in RDF/XML. Ed. Dave Beckett et al. 31 Jul 2001. University of Bristol. 01 Jun 2004
http://www.dublincore.org/documents/dcmes-xml/

[WAKA] "waka: A replacement for HTTP." Roy Fielding. Nov 2002. ApacheCon. Presentation by Roy Fielding on WAKA, which was designed from the ground up to be REST-ful. This presentation also

highlights some ways in which the HTTP header model violates REST, including mandatory extensions, scoped headers, and separating different types of headers.
http://www.gbiv.com/protocols/waka/

[AsTMa?] AsTMa* Topic Map Engineering. Ed. Robert Barta. 2003
http://topicmaps.bond.edu.au/docs/25?style=printable and AsTMa*
http://astma.it.bond.edu.au/

[Tolog] tolog: A Topic Map Query Language. Ed. Lars Marius Garshol. Ontopia. 04 Jul 2004
http://www.ontopia.net/topicmaps/materials/tolog.html

[SNAPI] "Semantic Network API Summary." SourceForge.Net. 2004. Open Source Development Network. 1 Jun 2004.
http://sourceforge.net/projects/snapi

[TMRDF] Living with Topic Maps and RDF. Ed. Lars Marius Garshol. Ontopia. 04 Jul 2004
http://www.ontopia.net/topicmaps/materials/tmrdf.html.

[RDF&TM] Ten Theses on Topic Maps and RDF. Ed. Steve Pepper. Jun 2000. Ontopia. Aug 2002
http://www.ontopia.net/topicmaps/materials/rdf.html

[Integration of TM&RDF] On the integration of Topic Maps and RDF by Martin Lacher & Sefan Decker.
http://www.semanticweb.org/SWWS/program/full/paper53.pdf

[rest-discuss] "REST Discussion Mailing Group." Yahoo.Com. 13 Nov 2001. Yahoo! Inc. 2004
http://groups.yahoo.com/group/rest-discuss/

[REST Wiki] "REST wiki". Ed. by Mark Baker et alia. The REST wiki provides an open ongoing collaborative discussion on REST Architecture.
http://rest.blueoxen.net/cgi-bin/wiki.pl

[1060.org] "NetKernel." 1060.org: Netkernel Community Resources. 2002. 1060 Research Limited. 2004
http://www.1060.org

[well-formed-web] The Well-Formed Web - Exploring the limits of XML and HTTP. 2003. BitWorking, Inc.
http://www.wellformedweb.org

[Atom] The Atom Syndication Format. Ed. M. Nottingham. Dec 2003. MNot.net. 8 Dec 2003. Atom is a successor to RSS that is being standardized at the IETF.
http://www.mnot.net/drafts/draft-nottingham-atom-format-00.html,
http://www.intertwingly.net/wiki/pie/FrontPage,
http://www.atomenabled.org/.

[RSS2.0] Real Simple Syndication. Ed. George Matesky and Betty Guernsey. 07 Sep 2002. Spartanburg Herald Journal. 12 Jul 2004
http://blogs.law.harvard.edu/tech/rss

[xml:id] xml:id Version 1.0. Ed. Jonathan Marsh and Daniel Veillard. 2004. World Wide Web Consortium. 07 Apr 2004
http://www.w3.org/TR/xml-id/

[xmlIDsemantics-32] How should the problem of identifying ID semantics in XML languages be addressed in the absence of a DTD?
http://xml.coverpages.org/ni2004-04-13-a.html,
http://www.w3.org/2001/tag/doc/xmlIDsemantics-32.html

[TMGraphInXML] A Topic Maps Graph in XML. Ed. Michel Biezunski and Steven R. Newcomb. 27 Jun 2001
http://www.topicmaps.net/simpleTMGraph3.htm

[TMQL] Topic Map Query Language (TMQL). IsotopicMaps.org. 21 Mar 2004
http://www.isotopicmaps.org/tmql/

[OWL] OWL Web Ontology Language. Ed. Dan Connolly. 2004. World Wide Web Consortium. 15 Jun 2004
http://www.w3.org/2001/sw/WebOnt/

[RDQL] RDQL - A Query Language for RDF. Ed. Andy Seaborne. Jan 2004. World Wide Web Consortium. 09 Jan 2004
http://www.hpl.hp.com/semweb/rdql.htm
http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/

[URIQA] The URI Query Agent Model. Ed. Patrick Stickler. Nokia Corporation. 2003-2004
http://sw.nokia.com/uriqa/URIQA.html

[SeRQL] "The SeRQL Query Language." Ed. Aduna B.V. et al. 2002. World Wide Web Consortium. 06 Jul 2004.
http://www.openrdf.org/doc/users/ch05.html

[DIG] DIG (DL Implementors Group). Ed. Peter F. Patel-Schneider.
http://dl.kr.org/dig/ - The WWW pages of the DL Implementors Group.

[DIG Interface] "DIG Interface". Ed. Daniele Turi. 2004.
http://sourceforge.net/projects/dig - An open-source project developing a standardised XML interface to Description Logics systems developed by the DL Implementation Group.
