DAP Web Services Specification
DRAFT

James Gallagher* Nathan Potter† John Chamberlin‡

2005-11-02
Revision: 12525

1 Introduction

The purpose of this paper is to outline different web service design choices being considered by OPeNDAP, to provide a rationale for their different characteristics and to solicit comments about them.

The phrase Web services is used to describe a range of network-based and distributed computing systems. Two characteristics common to virtually all such systems are that clients and servers on different computers interact to perform some computation and that the World Wide Web infrastructure supports that interaction.

Two classes of web services have emerged. The first type of web service system uses the REST architecture where *resources* are represented using URI and HTTP is used to return the current *representation* of that resource. These systems are based on HTTP, XML and URI and is described by Fielding and others in a variety of documents.[2][3] In these systems, XML is typically used to encode the representation (although for the web in general, HTML is).[1] Such systems are also often called 'network-based' because the existence of network operations is made explicit.

The second type of system follows the Remote Procedure Call model and uses the Simple Object Access Protocol (SOAP) along with other technologies such as WSDL and UDDI and builds a distributed system (where the network operations are hidden from client programs). Unlike HTTP/URI systems, using SOAP/WSDL/UDDI it is possible to build a system where clients can access machine readable interface descriptions (using WSDL records which can themselves be located using UDDI) and then use the RPCs in those descriptions to access a server. SOAP and its related protocols and technologies are described in great detail by the W3C as well as in numerous books.

It's important to note that REST describes an architecture, as does RPC. SOAP, HTTP, et c., are technologies which can be used to implement systems designed using one of these architectures.

These two approaches to building systems both have strong points. In the past, RPC-like systems have not scaled well beyond organizational boundaries, but SOAP may provide an additional infrastructure which allows it to overcome this limitation. At the same time the HTTP/URI based systems have been successful to

*OPeNDAP, Inc. jgallagher@opendap.org
†Oregon State University, ndp@coas.oregonstate.edu
‡OPeNDAP, Inc. jchamberlain@OPeNDAP.org
[1] XML or HTML are not the only syntaxes, but they are the standard ones described in the bulk of the documentation.

date but all require 'hand coding' of clients. This appears to be an inherent characteristic of such a system, but that also may not be a drawback because completely automated discovery and use of web services using UDDI/WSDL/SOAP may not be realistic when dealing with the typical Earth-science data objects.

We provide more information on the SOAP/WSDL technology and the REST architecture in Appendices A and C of this paper.

The remainder of this paper describes a SOAP messaging interface and a HTTP/GET interface for the OPeNDAP server. In both cases we break the server's interface down into two groups of services: Foundation Services and Presentation Services.

2 Foundation Services implemented using SOAP

The core services in the OPeNDAP hierarchy are called the *Foundation Services*. There are three of these services: **GetDDX**; **GetData**; and **GetBlob**. The **GetDDX** and **GetData** services are based on SOAP messaging.[2] The **GetBlob** service does not use SOAP. Instead it provides access to a serialized binary encoded data stream in an out-of-band manner.

The *Foundation Services* are all exposed through the single service name: *OPeNDAP*. This was done, in part, to allow clients to pool requests. A client may send a single message to the *OPeNDAP* service that contains multiple requests (see Section 2.5 on page 9).

2.1 GetDDX Service

The **GetDDX** service is used to provide client access to the DDX representation of a *Dataset* (See The Data Access Protocol—DAP 2.0). The client must send a SOAP message containing a valid constraint expression [3] to the service. In response the service will return the DDX of the *Dataset*, constrained as indicated in the constraint expression. This DDX will contain the message and service URL for the **GetBlob** service call that will return the Blob data. This service is the functional equivalent of the old DODS URL `.dds` response but note that the DDX also includes all of the information that was separated in the DAS response in DAP2, so it also provides the same information that DAP2 provides using the `.das` response.

For example, the SOAP body of a **GetDDX** message might look like:

```
<soap:Body>
    <GetDDX name="data.set.name">
        <Constraint>
            .
            .
            .
        </Constraint>
    </GetDDX>
</soap:Body>
```

[2] . . . as opposed to SOAP RPC. See Appendix A.

[3] As is the case with DAP2, a valid constraint expression is the empty string, which provides no constraint on the returned data (i.e., it instructs the server to return all of the data in the data source).

The SOAP body of the returned message might look like:

```
<soap:Body>
    <Dataset name="data.set.name">
        <Array name="sst">
            .
            .
            .
        </Array>
        <BlobRequest URL="http://hostname/OPeNDAP/Blob/">
            <GetBlob name="data set pathname">
                <Constraint>
                    .
                    .
                    .
                </Constraint>
            </GetBlob>
        </BlobRequest>
    </Dataset>
</soap:Body>
```

2.1.1 WSDL for the GetDDX service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="GetDDX"
   targetNamespace="http://xml.opendap.org/wsdl/getDDX.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://xml.opendap.org/wsdl/GetDDX.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <message name="GetDDXRequest">
      <part name="data_source" type="xsd:string"/>
      <!-- It's really an XML fragment... -->
      <part name="constraint" type="xsd:string"/>
   </message>
   <message name="GetDDXResponse">
      <!-- Is this the best way to describe a returned XML document? -->
      <part name="DDX" type="xsd:string"/>
   </message>

   <portType name="DDX_PortType">
      <operation name="getDDX">
         <input message="tns:GetDDXRequest"/>
         <output message="tns:GetDDXResponse"/>
      </operation>
   </portType>

   <binding name="DDX_Binding" type="tns:DDX_PortType">
      <soap:binding style="document"
                    transport="http://schemas.xmlsoap.org/soap/http" />
      <operation>
         <soap:operation soapAction="http://test.opendap.org/opendap"/>
         <input>
            <soap:body use="literal"/>
         </input>
         <output>
            <soap:body use="literal"/>
         </output>
      </operation>
   </binding>

   <service name="OPeNDAP">
      <documentation>WSDL File for OPeNDAP SOAP Web Services</documentation>
        <port name="ddx" binding="tns:DDX_Binding">
           <http:address location="http://test.opendap.org/opendap/"/>
        </port>
   </service>
</definitions>
```

2.2 GetData Service

The **GetData** service is used to provide client access (via SOAP) to the DDX representation of a *Dataset* (see DAP Objects paper) bundled with the Blob that contains the serialized binary data content of the returned DDX. That is, this service provides a 'SOAP way' to get the data values. This paper also describes an out-of-band data access 'pseudo-service' which has some distinct advantages over this service (See Section 2.4 on page 8). The client must send a SOAP message containing a valid constraint expression to the service. In response the service will return the DDX of the *Dataset*, constrained as indicated in the constraint expression. This will be followed, in the same message, by a `BlobData` element whose content is the Base64[4] encoded serialized binary data content of the returned DDX constrained as specified in the constraint expression. As always, the DDX will contain the both the service URL for the **GetBlob** service call and the message to send to the service URL that will return the Blob data. This service is the functional equivalent of the old DODS URL `.dods` response.

An example call to the **GetData** service might look like:

```
<soap:Body>
    <GetData name="data.set.name">
        <Constraint>
            .
            .
            .
        </Constraint>
    </GetData >
</soap:Body>
```

The SOAP body of the returned message might look (with liberties taken with the XML formatting for readability) something like:

```
<soap:Body>
    <Dataset name="data.set.name">
        <Array name="sst">
            .
            .
            .
        </Array>
        <BlobRequestMsg SrvcURL="hostname:12001/OPeNDAP/BSrvc">
            <GetBlob name="data.set.name">
                <Constraint>
                    .
                    .
                    .
                </Constraint>
            </GetBlob>
        </GetBlobRequestMsg>
    </Dataset>
    <BlobData>
    7xUzYGtGWgAAABkAAAAkVGhpcyBpcyBhIGRhdGEgdGVzdCBzdHJpbmcgKHBhc3MgMCk
    uAAAAJFRoaXMgaXMgYSBkYXRhIHRlc3Qgc3RyaW5nIChwYXNzIDEpLgAAACRUaGlzIGl
    zIGEgZGF0YSB0ZXN0IHN0cmluZyAocGFzcyAyKS4AAAAkVGhpcyBpcyBhIGRhdGEgdGV
    zdCBzdHJpbmcgKHBhc3MgMykuAAAAJFRoaXMgaXMgYSBkYXRhIHRlc3Qgc3RyaW5nICh
    wYXNzIDQpLgAAACRUaGlzIGlzIGEgZGF0YSB0ZXN0IHN0cmluZyAocGFzcyA1KS4AAAA
    kVGhpcyBpcyBhIGRhdGEgdGVzdCBzdHJpbmcgKHBhc3MgNikuAAAAJFRoaXMgaXMgYSB
    kYXRhIHRlc3Qgc3RyaW5nIChwYXNzIDcpLgAAACRUaGlzIGlzIGEgZGF0YSB0ZXN0IHN
    0cmluZyAocGFzcyA4KS4AAAAkVGhpcyBpcyBhIGRhdGEgdGVzdCBzdHJpbmcgKHBhc3M
    gOSkuAAAAJVRoaXMgaXMgYSBkYXRhIHRlc3Qgc3RyaW5nIChwYXNzIDEwKS4AAAAAAAA
    lVGhpcyBpcyBhIGRhdGEgdGVzdCBzdHJpbmcgKHBhc3MgMTEpLgAAAAAAACVUaGlzIGl
    IGEgZGF0YSB0ZXN0IHN0cmluZyAocGFzcyAxMikuAAAAAAAAJVRoaXMgaXMgYSBkYXRh
    </BlobData>

</soap:Body>
```

Although the **GetData** service provides a full SOAP based interface to data values stored in an OPeNDAP server, it is strongly recommend that clients combine the use of the **GetDDX** service and the **GetBlob** service instead. Since the **GetData** service uses SOAP to carry binary data content the Base64 encoding of the data is necessary to allow the data to be transmitted in an XML document. This has (at least) two unfortunate ramifications: First, the number of bytes transmitted is increased by a factor of 33 percent. Secondly, the entire response must be sent in the document, which eliminates the possibility of streaming the response to the client so that the client can process it on the fly.

2.3 Alternatives to Base64 Encoding

An alternative to providing data using Base64 encoding is to use SOAP with Attachments (SwA).[5] This provides a way to bundle binary data without incurring the size penalty of Base64 encoding. Unfortunately, SwA, like Base64 encoding, does not support data streaming.

Other documents that relate to SOAP with Attachments are the XML-binary Optimized Packaging[7] (XOP) and SOAP Message Transmission Optimization Mechanism[6] (SMTOM). At one point Microsoft and others were developing a MIME-like document format which they called DIME that could support the SwA solution *and* streaming responses. However, the references to DIME from Microsoft now describe this technology as superseded by SMTOM. It's not clear if SMTOM and XOP together will support streamed responses. It's also important that these rely on SOAP 1.2, which may have poor support in a wide variety of languages for some time.

It seems that, using the current technology, SwA improves on Base64 encoding even though it does not suppport streaming responses. It saves on the chore of performing the encoding and decoding, does not incur the size penalty of Base64 and the (encoded) data values are not processed by the XML parser.

Here's the previous response using SOAP with Attachments:

```
MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary; type=text/xml;
        start="<data.set.name.blob@hostname>"
Content-Description: This is the optional message description.

--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <data.set.name.blob@hostname>

<?xml version='1.0' ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
    <Dataset name="data.set.name">
        <Array name="sst">
            .
            .
            .
        </Array>
        <BlobRequestMsg SrvcURL="hostname:12001/OPeNDAP/BSrvc">
            <GetBlob name="data.set.name">
                <Constraint>
                    .
                    .
                    .
                </Constraint>
            </GetBlob>
        </GetBlobRequestMsg>
    </Dataset>
    <BlobData href="cid:data.set.name.blob@hostname"/>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

--MIME_boundary
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary
Content-ID: <data.set.name.blob@hostname>

...binary XDR encoded data...
--MIME_boundary--
```

2.4 GetBlob Service

The **GetBlob** service is NOT a SOAP interface. This foundation service is used by clients to retrieve the serialized binary content of a *Dataset*. To invoke this service a client connects to the correct port on an OPeNDAP

server and sends an XML message to the server. The OPeNDAP server replies with the serialized binary data content of the *Dataset* described in the request as described in the DAP Objects paper.

It is important to provide a way for the receiver of the **GetBlob** response to discern error messages that might appear at any point in the stream, since the response returned by **GetBlob** is not a contained within a SOAP response, but instead is a stream of data values. Suppose, for example, that several variables are to be returned and the server (i.e., the **GetBlob** response generator) reads and serializes one, then the next and so on. An error accessing or returning the data for the second variable might not be seen until the values for the first have already been sent. To accommodate such a situation, the stream returned by **GetBlob** will be 'chunked,' in a way similar to HTTP/1.1 chunked responses, so that a premature end of data and the start of information about an error (e.g., an ErrorX response) can be detected.

An example **GetBlob** message might look like:

```
<GetBlob name="data set pathname">
    <Constraint>
        .
        .
        .
    </Constraint>
</GetBlob >
```

In response the server will return a multi-part MIME document which contains a brief description of the source of the data and the data themselves, encoded using XDR.[4]

2.5 Pooled Requests

The *OPeNDAP* service supports pooling requests. A client may collect multiple **GetDDX** and **GetData** service requests and send them to a server in a single request. The server will return the appropriate items in the same order that they were requested. All of the SOAP based *Foundation Services* services may be pooled. The **GetBlob** service call MAY NOT be pooled.

For example, a client could request 3 DDX's in a single pooled request:

[4]Proposed is an optimization of the 'always use XDR' approach where the client announces to the server that it uses either big- or little-endian representation and the server then responds either using the XDR-based big-endian (as default) or matches the client's representation, thus eliminating, in the worst case scenario, the need to transform the data values on both ends.

```
<soap:Body>
    <GetDDX name="data.set.01">
        <Constraint/>
    </GetDDX>
    <GetDDX name="data.set.02">
        <Constraint/>
    </GetDDX>
    <GetDDX name="data.set.03">
        <Constraint/>
    </GetDDX>
</soap:Body>
```

And the server would return:

```
<soap:Body>
    <Dataset name="data.set.01">
        .
        .
        .
    </Dataset>
    <Dataset name="data.set.02">
        .
        .
        .
    </Dataset>
    <Dataset name="data.set.03">
        .
        .
        .
    </Dataset>
</soap:Body>
```

The pooled requests need not be homogeneous. Consider:

```
<soap:Body>
    <GetDDX name="data.set.01">
        <Constraint/>
    </GetDDX>
    <GetData name="data.set.02">
        <Constraint/>
    </GetDDX>
    <GetDDX name="data.set.03">
        <Constraint/>
    </GetDDX>
    <GetBlobData name="data.set.04">
        <Constraint/>
    </GetBlobData >
</soap:Body>
```

And the server would return:

```
<soap:Body>
    <Dataset name="data.set.01">
        .
        .
        .
    </Dataset>
    <Dataset name="data.set.02">
        .
        .
        .
    </Dataset>
    <BlobData>
        .
        .
        .
    </BlobData>
    <Dataset name="data.set.03">
        .
        .
        .
    </Dataset>
    <BlobData name="data.set.04">
        .
        .
        .
    </BlobData>

</soap:Body>
```

3 Presentation Services using SOAP

The Presentation services, such as DAP2's Info response, will be implemented using SOAP RPC. These higher level services provide the kind of functionality that will allow our users to use UDDI as a discovery mechanism, and should be kept in the SOAP RPC domain.

Presentations Services to be described: Info, Version, HTML (a form interface).

Note that the development of 'Server4' will introduce software which uses THREDDS to encode directories and catalogs in XML. I assume that a SOAP RPC interface will be developed for these and so they will fall into the Presentation Services.

The Presentation Services are responses which the OPeNDAP servers will support that are not, strictly speaking, DAP responses.

4 Foundation Services Using HTTP/GET

The current server produced by OPeNDAP provides a HTTP/GET interface for the *Foundation Services* (and the Presentation services, too). Different responses are returned by adding a suffix to the data source URI. This interface has been documented extensively elsewhere so, rather than repeat that in this paper we will describe how the DAP4 features will alter this already-documented interface.

The DAS and DDS responses are combined in one document in DAP4. This new response is called the DDX. In the DDX each variable's attributes are integral to the variable itself. Data sources themselves can still have attributes independent of the variables' attributes (i.e., global attributes) and, as with DAP2's DAS, there can be any number of additional attribute containers.

In addition to the DDX replacing the DAS/DDS pair, DAP4 bundles the *BLOB URI* in the DDX. Every DDX contains the BLOB URI which references the data described by that DDX. Of course, clients are under no obligation to dereference these BLOB URI, but each DDX response will now contain all the information needed to access the data it describes. When a DDX is requested using a *Constraint Expression* (CE) the BLOB URI returned with the DDX response will return those values as constrained by the CE.

The DDX replaces the DAS/DDS and also the DataDDS responses. At the same time the DDX response also frees the client from remembering how to form a URI for data since each DDX contains the BLOB URI which references the data described by the DDX. This design also frees the server from using the same URI base for both the metadata (DDX) and data (BLOB URI). In fact the BLOB URI might reference data stored on a completely different machine.[5]

The adoption of XML as the syntax for the DDX frees clients and servers from relying on parsers specific to the DAP, at least in part. Because the DAP is typically used to describe fairly complex data sources, most of the potential clients will need some form of DAP-specific knowledge.

NOTE: The OPeNDAP Data Server version 3.5 supports an experimental version of the DDX response. Clients, including (or maybe especially) web browsers, can access this by appending the suffix `.ddx` to a DAP URL passed to a browser. The OPeNDAP test data site will support this once it's online. Its URL is: `http://test.opendap.org/OPeNDAP-3.5/nph-dods/data/`. From there you can navigate to a data source and replace the `.html` suffix with .ddx. In addition, `libdap` version 3.5.3 contains client side tools to use the DDX response.

4.1 HTTP/GET and the Request Syntax

In the HTTP/GET interface, the query string is still used to pass the CE from the client to the server. The disadvantage to this is that the current CE requires a fairly sophisticated parser while simpler uses of the query string use a parameter-value pair which is almost trivial to read. We will investigate using the more common query string syntax with the DAP4 servers.[6] This should avail DAP4 CEs some of same benefits that the DDX will get from use of XML; that off the shelf toolkits will be able to parse the CEs.

[5] The SOAP interface described before is also capable of this.

[6] For the sake of backward compatibility, we will support the DAP2 CE grammar in DAP4.

4.2 WSDL and the HTTP/GET interface

The HTTP/GET interface can be described using WSDL in a way very similar to the SOAP interface. See Web Services Description Language (WSDL) 1.1[1] for more information.

Here is an example WSDL description for the DDX service:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="DDXService"
   targetNamespace="http://xml.OPeNDAP.org/wsdl/DDXService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://xml.OPeNDAP.org/wsdl/DDXService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <message name="DDXRequest">
      <part name="data_source" type="xsd:string"/>
      <part name="proj" type="xsd:string"/>
      <part name="sel" type="xsd:string"/>
   </message>
   <message name="DDXResponse">
      <!-- Is this the best way to describe a returned XML document? -->
      <part name="DDX" type="xsd:string"/>
   </message>

   <portType name="DDX_PortType">
      <operation name="getDDX">
         <input message="tns:DDXRequest"/>
         <output message="tns:DDXResponse"/>
      </operation>
   </portType>

   <binding name="DDX_Binding" type="tns:DDX_PortType">
      <http:binding verb="GET"/>
       <operation name="getDDX">
           <!-- two options below, one commented out -->
           <http:operation location="nph-dods/(data_source).ddx?proj=(proj)&sel=(sel)"/>
           <!-- http:operation location="nph-dods/(data_source)/ddx/(proj)/(sel)"/ -->
           <input>
               <http:urlReplacement/>
           </input>
           <output>
               <!-- Should this be application/xml? -->
               <mime:content type="text/xml"/>
           </output>
       </operation>
   </binding>

   <service name="DDX_Service">
      <documentation>WSDL File for DDXService</documentation>
        <port name="ddx" binding="tns:DDX_Binding">
           <http:address location="http://test.opendap.org/opendap/"/>
        </port>
   </service>
</definitions>
```

A Web Services and SOAP

SOAP is used in two basic ways, as a tool for doing RPC on remote systems, and as a messaging service.

When SOAP is used to perform RPC the parameters passed and the return values must be consistent with the SOAP data model (unless custom encodings are used, more on that later). This model is relatively simplistic, it contains a collection of atomic types, arrays, and structures. In the SOAP data model values must be associated with each instance of an atomic type, this includes members of an array or structure.

Given the type constraints and the embedded value requirements of the SOAP data model it is not practical to try to shoe-horn the OPeNDAP data model directly into a SOAP data model representation model.

In order pass our DAP objects via SOAP methods we have 2 choices: Extend the SOAP data model to cover our custom data types (so that our types can be used in an RPC request) or rely on SOAP messaging.

Custom data types require the use of custom serialization and de-serialization classes that must be integrated into the SOAP server (and client). These classes are tightly coupled to the particular implementation of the SOAP engine (such as Apache AXIS, GLUE, etc.)

Messaging allows us to pass W3C DOM trees back and forth between server and client. Each side can then act upon them as necessary (parse them into a Java/C++ class in memory, extract pertinent information directly from the DOM elements, what have you)

Using a SOAP messaging scheme will cut us off from the automatic WSDL generation tools that have been developed for the SOAP RPC model. However, this may not be such a large issue. Ultimately, having a custom data type in an RPC call only buys you the information (at the WSDL level) that this method requires (for example) a ”DDS” or a ”ConstraintExpression” but probably doesn’t lead you to an implementation of such a beast.

B Ethan Davis Had this comment:

I like the single ’OPeNDAP’ service idea with the requested foundation service encoded in the request message. I wonder if the DDX should contain a more generic request message rather than one specific for the Blob, e.g.,

```
</Dataset>
    ...
    <RequestMsg SrvcURL="http://hostname/axis/services/OPeNDAP"
                dataSetName="data.set.name">
            <Constraint>
                ...
            </Constraint>
    </RequestMsg>
</Dataset>
```

That way it doesn’t limit the type of request a user can build from the DDX info (or rather, it doesn’t give the appearance of limiting the users request).

C REST: Representational State Transfer

This summary of REST was copied from webservices.sml.com[7] *I've added my own editorial comments (in italics) where they seem relevant. jhrg.*

REST is a model for distributed computing. It is the one used by the world's biggest distributed computing application, the Web. When applied to web services technologies, it usually depends on a trio of technologies designed to be extremely extensible: XML, URIs, and HTTP. XML's extensibility should be obvious to most, but the other two may not be.

The acronym 'REST' was coined by Roy Fielding in his Ph. D. Dissertation.[3] Fielding describes REST as "a hybrid style derived from several . . . network-based architectural styles . . . combined with additional constraints that define a uniform connector interface."[8] *He also points out that most (previously) published information on software architecture excludes data as an "important architectural element"*[9] *This fits nicely with the discussion below of URIs used to refer to 'resources' which are typically data elements.*

URIs are also extensible: there are an infinite number of possible URIs. More importantly, they can apply to an infinite number of logical entities called "resources." URIs are just the names and addresses of resources. Some REST advocates call the process of bringing your applications into this model "resource modeling." This process is not yet as formal as object oriented modeling or entity-relation modeling, but it is related.

The strength and flexibility of REST comes from the pervasive use of URIs. This point cannot be overemphasized. When the Web was invented it had three components: HTML, which was about the worst markup language of its day (other than being simple); HTTP, which was the most primitive protocol of its day (other than being simple), and URIs (then called URLs), which were the only generalized, universal naming and addressing mechanism in use on the Internet. Why did the Web succeed? Not because of HTML and not because of HTTP. Those standards were merely shells for URIs.

Fielding places significant emphasis on URI, stating that the architecture's goals are ". . . achieved by placing constraints on connector semantics where other styles have focused on component semantics."[10] *He also states that URI " are both the simplest element of the Web architecture and the most important."*[11] *They map the concept represented by a resource to the concrete representation of that resource at a given time. REST defines "a resource to be the semantics of what the author intends to identify, rather than the value corresponding to those semantics at the time the reference is created."*[12] *Late binding of references is crucial for cross-organizational systems.*

HTTP's extensibility stems primarily from the ability to distribute any payload with headers, using predefined or (occasionally) new methods. What makes HTTP really special among all protocols, however, is its built-in support for URIs and resources. URIs are the defining characteristic of the Web: the mojo that makes it work and scale. HTTP as a protocol keeps them front and center by defining all methods as operations on URI-addressed resources.

[7] http://webservices.xml.com/pub/a/ws/2002/02/20/rest.html
[8] Fielding, p 76.
[9] Fielding, p 23.
[10] Fielding, p 148.
[11] Fielding, p 109.
[12] Fielding, p 111.

The most decisive difference between web services and previous distributed computing problems is that web services must be designed to work across organizational boundaries. Of course, this is also one of the defining characteristics of the Web. This constraint has serious implications with respect to security, auditing, and performance.

REST first benefits security in a sort of sociological manner. Where RPC protocols try as hard as possible to make the network look as if it is not there, REST requires you to design a network interface in terms of URIs and resources (increasingly XML resources). REST says: "network programming is different than desktop programming – deal with it!" [*This idea resurfaces every so often. The first place I (jhrg) saw it was in a paper about Sun's RPC technology written by a group within Sun.[8] Fielding also cites a reference to Tanenbaum and van Renesse that makes the same point when describing the distinction between 'network-based' and 'distributed' systems (that the former is not transparent to the user while the latter is).*[13]] Whereas RPC interfaces encourage you to view incoming messages as method parameters to be passed directly and automatically to programs, REST requires a certain disconnect between the interface (which is REST-oriented) and the implementation (which is usually object-oriented).

> *What makes HTTP [in the text, HTTP/1.1 is described as being designed using REST. jhrg] significantly different from RPC is that the requests are directed to resources using a generic interface with standard semantics that can be interpreted by intermediaries almost as well as by the machines that originate services. The result is an application that allows for layers of transformation and indirection that are independent of the information origin, which is very useful for an Internet-scale, multi-organization, anarchically scalable information system.*[14]

References

[1] Erik Christensen, Fancisco Curbera, Greg Meredth, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[2] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol — HTTP/1.1. RFC 2616.

[3] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[4] N. Freed and N. Borenstein. Multipurpose internet mail extensions (MIME) part one: Format of internet message bodies. http://www.ietf.org/rfc/rfc2045.txt, 1996.

[5] W3C. Soap messages with attachments. http://www.w3.org/TR/SOAP-attachments, 2000.

[6] W3C. SOAP message transmission optimization mechanism. http://www.w3.org/TR/soap12-mtom/, 2005.

[7] W3C. XML-binary optimized packaging. http://www.w3.org/TR/xop10/, 2005.

[8] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing, 1993.

[13] Fielding, p 24.
[14] Fielding, p 142.