Hold Your Nose vs. Follow Your Nose

Observations on the state of service description on the Web

Mike Amundsen
Principal API Architect, Layer 7 Technologies
mca@amundsen.com

ABSTRACT

The first service description format for the Web appeared in 2001. The second didn't arrive until 2007. In 2010 there were less than five known service description formats. As of this writing there are more than a dozen. We're experience a "Cambrian explosion" of service description formats!

Why all these new formats? Is there something missing that each new type tries to fix? Or are these new formats just repeating the same patterns over and over? This paper 1) explores two main approaches to service description design (describing functionality and describing things), 2) reviews key shortcomings of existing approaches that lead to client applications tightly bound to a single service instance, 3) offers up a new set of measures for useful service description, and 4) introduces an alternative format that supports the key Web principle of *follow your nose*.

1. BRIEF TOUR OF SERVICE DESCRIPTION FORMATS

The first service description format for the Web appear in 2001. The second one didn't arrive until 2007. By 2010 there were less than five known service description formats. As of this writing there are more than a dozen. We're experience a "Cambrian explosion" of service description formats!

Despite the wide number of formats available, they all fall into two rather narrow categories: those that describe *functionality* and those that describe *things.* Both approaches have roots in early computer programming models such as interface description language (IDL)[1] and object-oriented programming(OOP) and neither approach is appropriate for the Web. The Web was conceived "a web of nodes in which the user can browse at will."[2]. It is meant to be a place where items are linked together in a myriad of ways and where the notion of "follow your nose"[4] means users (and even machines) should be able to travel any desired path that may exist within the network. Service description formats today do not support these notions. In fact most of them are openly hostile to these notions. As a result, most *client* implementations derived from these service documents are unable to travel desired paths and are instead designed to "hold their nose" and ignore any additional links or data elements that may appear over time.

In this paper, the first section focuses on two main approaches to service description: describing functionality and describing things. It contains a handful of examples and points to problems with these common approaches. Section 2 (Shortcomings of Existing Formats) details the problems in depth. Section 3 identifies new measures for a successful format that supports and promotes the principles of the Web (Toward a New Service Description Format), and finally, Section 4 presents a possible solution (Application-Level Profile Semantics (ALPS)) along with some critical commentary.

WS-REST 2014, Seoul, Korea
ACM (to be determined).

1.1 Describing Functionality

One class of service descriptions focuses on capturing *functionality* by selecting portions of internal programming code and exposing it for use on the Web. The most well-known instance of this approach is Web Services Definition Language or WSDL[3] for SOAP Services[5] . Essentially, WSDL describes remote procedure calls, objects, and data types that appear in the source code of the service being described.

```
<binding name="StockQuoteSoapBinding"
  type="tns:StockQuotePortType">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetLastTradePrice">
    <soap:operation
      soapAction="http://example.com/GetLastTradePrice"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
```

A similar approach is used by RESTdesc[22]. Described as "a semantic semantic service description approach that centers on functionality"[21], RESTdesc uses the RDF/Notation3 format to describe pre-conditions, state transfer requirements, and expected return values (post-conditions).

```
@prefix : <http://example.org/image#>.
@prefix http: <http://www.w3.org/2011/http#>.
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.
{
  ?image :smallThumbnail ?thumbnail.
}
=>
{
  _:request http:methodName "GET";
            http:requestURI ?thumbnail;
            http:resp [ http:body ?thumbnail ].
  ?image dbpedia-owl:thumbnail ?thumbnail.
  ?thumbnail a dbpedia:Image;
             dbpedia-owl:height 80.0.
}.
```

Microsoft's Conceptual Schema Definition Language (CSDL)[34] takes a similar approach by exposing functions via the `<FunctionImport name="...">` element within a schema that includes `EntityType`, `Association` and other similar elements.

```
<!-- Service Operation -->
<FunctionImport Name="CustomersByCity"
    EntitySet="Customers"
    ReturnType="Collection(SampleModel.Customer)"
    m:HttpMethod="GET">
  <Parameter Name="city" Type="Edm.String" Mode="In"/>
</FunctionImport>
....
```

It should be noted that exposing functions via RPC-style IDL run counter to the hyperlink approach used in HTML[31], VoiceXML[32] and other early formats for the Web such as HyTime[23].

1.2 Describing Things

Another common approach is to use service description formats to define *things* or objects that can be passed between client and server. Usually these *things* can be manipulated via a set of protocol methods. The Atom Publishing Protocol[10] uses Service Documents to define `workspaces`, `collections`, and `categories` which are used to access `entry` objects which can be modified by using HTTP for "the creating, editing, and deleting of Entry and Media Resources."[10]

```
<?xml version="1.0" encoding='utf-8'?>
<service xmlns="http://www.w3.org/2007/app"
  xmlns:atom="http://www.w3.org/2005/Atom">
  <workspace>
    <atom:title>Main Site</atom:title>
    <collection
      href="http://example.org/blog/main" >
      <atom:title>My Blog Entries</atom:title>
      <categories
        href="http://example.com/cats/forMain.cats" />
     </collection>
  </workspace>
</service>
```

Many other service description formats take the same approach including (among others) the Web Application Description Language (WADL)[8], Swagger[9], JSON Home
[14].

The manipulation of these objects/things is almost always tied to the HTTP protocol and a subset of the available methods: `POST`, `GET`, `PUT`, and `DELETE`. It is likely no coincidence that this protocol subset closely matches a subset of SQL Data Manipulation Language (DML)[33]: `INSERT`, `SELECT`, `UPDATE`, and `DELETE`. It is common for service implementations to simply expose stored data tables (users, customers, products, etc.) over HTTP by defining resources with the same names and serializing data rows using XML or JSON. This manipulation pattern is often referred to as *CRUD* or Create, Read, Update, Delete.

Focusing on serializing objects between client and server via the CRUD model over HTTP ignores many of key design features of HTTP. The ability to transfer arbitrary state values is lost (unless new objects are invented), important HTTP methods are ignored (including HEAD and PATCH), and the freedom to treat message payloads as text documents (plain text), argument lists (`application/x-www-forms-urlencoded`), or even binary streams (PNG) is sidelined in favor of simple data manipulation and object-passing.

1.3 Summary

In one of his early proposals, Tim Berners-Lee described the WorldWideWeb as "...forming of a web of information nodes rather than a hierarchical tree or an ordered list..."[2]. Yet, most description formats today rely heavily on the notion of function libraries and object hierarchies. Legacy thinking based on early Interface Description Languages (IDL) was used to create the first widely-used service description format (WSDL) and the object oriented ethos of the 1980s continues to influence many other formats today (WADL, Swagger, etc.).

2. SHORTCOMINGS OF EXISTING FORMATS

The two approaches in service description formats - that of describing functionality and describing things - ignore fundamental aspects of the Web. They make it difficult to support the *follow-your-nose* principle and hard to support the notion of "serendipitous reuse"[24]. In some cases, these formats were created and standardized before the ideas like "a web of nodes" and "serendipitous reuse" were commonly understood. In other cases, service descriptions were explicitly designed to ignore features of the Web in order to continue existing models.

No matter the reasons, the bulk of service description formats today suffer from a number of shortcomings. These include 1) a focus on object-passing, 2) describing instances rather than classes of services, 3) specifying protocols andc4) constraining media type usage. What follows is a review these shortcomings along with some conclusions on how they affect the process of creating robust client applications for the Web.

2.1 Focused on Objects

Many description formats for the Web focus on explicitly defining objects that will be passed between client and server. For example, the WSDL format uses the `<type>` element which "...encloses data type definitions that are relevant for the exchanged messages."[3].

One of the key problems of focusing on object-passing for Web services is that is raises the bar for shared understanding between client and server. Clients and servers must agree *a priori* on the shape of each and every object to be shared. In addition, the actual shape of each object becomes harder and harder to agree upon as more parties being to use the object. As Fielding states, "it is far easier to standardize representation and relation types than it is to standardize objects and object-specific interfaces"[7]. There are many stories of IT projects foundering on the rocks of object hierarchy definition.

Object-focus may slow initial design and implementation of IT projects and can over-constrain interactions between parties at runtime. It also introduces additional fragility to the service since modifications in the shape and even the identity and meaning of complex objects is likely to change over time.

2.2 Limited to Instances

Almost all service descriptions - whether focused on functionality or things - are limited to decriging only a single instance of a service on the web. This means that client applications wishing to access the same functionality at two different servers will need to operate on two different description documents. This is true even when the description *format* for both servers is identical. In other words, service descriptions are not designed to describe a set or class of similar services operating indpendently at multiple locations.

This limitation occurs because most service description formats use static URLs as part of the service description. For example, the Web Application Description language (WADL)[8] employs a `<resources base="...">` element that defines the base URL for the service. This element has a set of child elements (`<resource path="...">`) which "describes a set of resources, each identified by a URI..."[8].

```
<resources base="http://example.com/">
  <resource path="widgets">
    <resource path="reports/stock">
      <param name="instockonly" style="matrix"
        type="xsd:boolean"/>
      ...
    </resource>
    <resource path="{widgetId}">
      ...
    </resource>
      ...
  </resource>
</resources>
```

Similarly, the Swagger[9] specification uses the `basePath` property and each member of the `apis` array contains a `path` property.

```
{
  "apiVersion": "0.2",
  "swaggerVersion": "1.2",
  "basePath": "http://petstore.swagger.wordnik.com/api",
  "resourcePath": "/pet"
  ...
  "apis":[
    {
      "path":"/pet/{petId}",
      "description":"Operations about pets",
      "operations":[
        {...}
      ]
    }
  ]
  "models" : {...}
}
```

Even though some of these formats mark the `path` element as optional, in practice every service publisher creates descriptions that include these values. As a result, each description constrains the list of possible resources by listing a *complete set* of URLs for the service instance. This practice has harmful side effects at runtime.

By offering up a list of static URLs, client applications are led into baking these URLs into their own code. This is a *feature* of code-generating tools for the WSDL format. Not only does this practice bind the client application to a single instance of the server, but it can also lead to broken client applications if the target server's base address or internal path URLs change over time.

Another problem with this approach is that client applications are bound to a closed set of interactions controlled only by the server. This means client applications wishing to interact with similar (but not identical) servers providing the same service (e.g. package tracking, shopping cart management, banking, etc.) are unable to easily move between providers.

Most service description formats today can only describe a single instance of a service and limit client's ability to interact with similar services on the Web. In this way client applications become *captives* of a single service instance.

2.3 Specifying Protocol

The majority of service description formats assume a single protocol for the service implementation and protocol is almost always HTTP. Very few formats allow providers to offer more than one protocol. For example, the WADL format "is designed to provide a machine process-able description of ... HTTP-based Web applications."[8]. In fact there is no indication of protocol within the message format itself. According to the XSD Schema for WADL[11] the `<method name="...">` element supports a limited and fixed set of values for the `name` property (GET, POST, PUT, HEAD, DELETE). It should be noted that all but one of these methods are also supported by the proposed Constrained Application Protocol (CoAP)[12].

Most of the recently created formats (e.g. Swagger[9], CSDL[34], JSON Home[14], etc.) assume HTTP as the only supported protocol. While this makes implementation initially simpler, it also reduces the reach and possibly the longevity of both the service and the service description format.

A notable exception to this "assumed HTTP" pattern is the WSDL[3] format. WSDL was designed from the start to support multi-ple protocol via the `<wsoap12:binding transport="..." />` element. The author was able to find documents detailing the use of SOAP over XMPP[15], Java Message Service[17], and even SOAP over SMTP[16].

Despite the existence of the protocol-agnostic WSDL format, the general pattern in creating service descriptions is to assume HTTP and to not support additional protocols.

2.4 Constraining Media-Types

Similar to the way they handle protocols, many existing service descriptions constrain the media type used at runtime to transfer state between client and server. In some cases (e.g. WSDL[3], Google Discovery Document[18], JSON Hyperschema[19], etc.) the format is simply assumed based on the format of the service description document (XML for WSDL and JSON for Google Discovery and JSON Hyperschema).

For some formats, the media type is defined per service. For example the RESTful Application Modeling Language (RAML)[20] uses a YAML format with a `mediaType` element at the API (service) level.

```
#%RAML 0.8
---
title: Example API
version: v1
mediaType: application/json
schemas:
  users: !include schemas/users.json
  user: !include schemas/user.json
resourceTypes:
... and so forth ...
```

Some service descriptions support describing the media types returned at the resource level. For example, WADL[8] has the `<representation mediaType="..." />` element.

```
<method name="GET">
 <request>
  <param name="format" style="query">
   <option value="xml" mediaType="application/xml"/>
   <option value="json" mediaType="application/json"/>
   </param>
   ...
  </request>
  <response>
   <representation mediaType="application/xml"/>
   <representation mediaType="application/json"/>
  </response>
</method>
```

The JSON Home[14] format uses the notion of "format hints" to provide information on possible media types used in responses.

```
{
 "resources": {
   "http://example.org/rel/widgets": {
     "href": "/widgets/"
   },
   "http://example.org/rel/widget": {
     ...
     },
     "hints": {
       "allow": ["GET", "PUT", "DELETE", "PATCH"],
       "formats": {
         "application/json": {}
       },
       ...
     }
   }
 }
}
```

The good news is that the pattern relying on a single media type in responses is not common in most of the new description formats.

2.5 Summary

Most of the existing service description formats exhibit shortcomings when it comes to supporting the Web principles of "follow you nose" and "serendipitous resuse." Chief among these are descriptions that focus on object-passing instead of state-transfer as a way of communicating domain information. All the reviewed formats define fixed URLs (absolute or relative) and, as a result are limited to describing a single instance of a service instead of a set or class of services that share a common description. Most formats assume a single transfer protocol is used (HTTP) and some still do not allow the media type of responses to be defined at either a service or resource level.

These constraints usually produce server implementations that are unable to share service descriptions among similar peers which are constrained to a single protocol and media type. As a result, client applications that can successfully interact with these services are usually bound tightly to that single service instance and are unable to accommodate new URLs that may appear over time and have no ability to select preferred protocols or media types when interacting with services. These are clients that are basically designed to memorize a single set of fixed interactions for a single service instance using a single protocol and media type. Any new information (new resources, new state transitions within existing resources, additional server instances, protocols, or formats) is ignored. These clients are built to *hold their nose* instead of *follow their nose* when it comes to interactions on the Web.

3. TOWARD A NEW SERVICE DESCRIPTION FORMAT

If existing formats fall short in describing services that support the follow-your-nose principle, what is needed instead? What would it take to create a description format that allows clients to follow their nose through the available nodes in the network? How can a service description promote serendipitous reuse? What is an alternative to describing functionality or describing things?

Below are some guiding principles to use when considering a new description format:

- It should not constrain URLs. Server implementors should be free to manage their own URL space as they see fit.
- It should not limit the protocols for an implementation of that service. Implementors should have the freedom to select the protocol that best fits their needs without having to create new description documents for the same service.
- It should not constrain the media types used to transfer state between client and server. Implementors should be allowed to select one or more media types that are appropriate for the target audience and technical goals.
- It should focus on defining a shared vocabulary of state and action elements understood by all parties and not dictate object models or hierarchies.
- It should clearly describe possible interactions (read, write, etc.) in machine-readable form.

By allowing servers to determine their own implementation details, while clearly defining both state values and actions in a machine-readable way, a single service description can be used in to build many different solutions and a single client application can still be expected to successfully interact with that service - even in cases where the client serendipitously locates a particular instance of that service for the first time.

What follows are additional details on the principles listed above.

3.1 Describe Affordances

First and foremost the service description must not define URLs or static resource model. Instead, the format should define affordances. These affordances should be machine readable and should fully describe the details of a client-server interaction including:

- Some indication of *why* someone (or some machine) would decide to use this affordance
- A list of any parameters that could be passed from client to server
- An indication of the protocol action to be used to initiate the request
- An indication of the possible values that will be returned when the request is completed

For example, below is an affordance description for searching on the Web

```
affordance:
  use      : search
  request  :
    action : read
    params : q
    format : application/x-www-form-urlencoded
  response :
    type   : links
    format : text/html
```

Table 1: Examples of Safety and Idempotence

Protocol	Method	Safe	Idempotent
HTTP	GET	Y	Y
	PUT	N	Y
	POST	N	N
FTP	RETR	Y	Y
	STOR	N	Y
	STOU	N	N

Notice that there is no URL or resource indication in the description. Service descriptions do not need to document URLs - they will be supplied at runtime by the server. Service descriptions should not define resources, either. They should focus on the actual actions that may be available within the problem domain (accounting, microblogging, etc.) and not constrain when or where those actions (affordances) will appear.

In some cases, a set of actions will only be available to administrators. In other cases, one service implementor may group affordances on a single resource and another may place one on each affordance on its own resource. Like URLs, these are decisions made at runtime by services and are not needed in the description

3.2 Focus on Vocabulary

Service descriptions that define object hierarchies greatly constrain both client and server implementations. Object trees are very dense models. Not only do object trees define data elements (`familyName`, `givenName`, `person`, `team`, etc.) they define the *relationships* between each data element (`familyName` and `givenName` are sub-elements of `person`, `person` is a member of a `team`, etc.). Also, in practical terms, the larger the audience you wish to share these definitions, the more difficult it is to gain agreement on the shape of object trees. This is even more problematic when the object trees themselves are large and/or the problem domain they describe (world health, economics, etc.) is large and/or complex.

Service documents should focusing on a shared vocabulary rather than a shared object model. Essentially, the service description should establish a *dictionary* of terms independent of their relationship to each other. The *relationships* can be expressed at runtime by the various services that use these agreed-upon terms.

By removing objects and relationships from the service document, the same description can support several different use cases. For example, a `team` might be assigned a `givenName`, etc. and there is no need to bake this into the service description.

3.3 Protocol Agnostic

Service descriptions should be designed to support multiple protocols (as does SOAP[5]) or leave protocol implementation out completely. Most service descriptions covered here are designed to support a single network protocol (HTTP) and refer to HTTP verbs such as `GET`, `POST`, etc. directly in the document. This renders the descriptions incomplete or invalid for implementations that wish to use CoAP[12], XMPP[26], and other protocols.

One way to decouple service description from protocol is to use an independent set of words for protocol verbs; ones that can easily be translated into protocol-specific verbs by implementors. A workable example was described by this author[27] based on characterizing network protocol requests as to their degree of “safety” and “idempotence” (see Table 1 for examples).

Service descriptions can use the concepts of safety and idempotence to accurately describe network request details without having to constrain the protocol used for implementing the service. This allows service descriptions to be useful across a wide range of protocols - even protocols not yet invented.

3.4 Media-Type Independence

Service description should either 1) allow each response to be defined with more than one media type or 2) describe responses in such a way that media type is unimportant for the description document and can be handled independently by implementors. This allows the same service description to be used no matter the message model (or models) used for implementing the service.

Some service documents constrain the media types used when constructing representations at runtime. This overly limits the use of the service description. There are some very nice features of some JSON-base service descriptions that are only designed to describe JSON representations. If a developer wants to render responses in XML, then some service descriptions are unacceptable. While it is possible to define the same service model in both XML and JSON, what of YAML? HTML? and any other format that may come along at some point? Do we continue to duplicate the same description model for each and every media type?

3.5 Summary

This section has explored a set of new goals for service description. Goals that do not over constrain implementors and still provide clear descriptions of data elements hypermedia affordances. With these goals in mind, it is possible to outline a new service description format that meets these new requirements.

4. APPLICATION-LEVEL PROFILE SEMANTICS (ALPS)

This section introduces the Application-Level Profile Semantics (ALPS)[28] service description format. This format was designed to support the goals outlined earlier in this paper including:

- Describe Interfaces, not an Implementations : Do not constrain the server’s URLs or resource model.
- Protocol Agnostic : Describe network interactions in terms of “safety” and “idempotence”
- Media-Type Independence : Allow service programmers to select the media types used in the implementation
- Focus on Vocabularies, not Object Models : Allow service descriptions to define data as “terms” which are independent of any implementation hierarchy or object model.

Along with these already stated goals, the ALPS format was design to also support:

- Runtime Identification of Profiles : Service document URLs are used with the “profile” link relation value[29] so that both parties can recognize and acknowledge shared understanding of the service description
- Documentation Generation : ALPS documents can also be used to provide machine-generated human-readable documentation
- Code Generation : ALPS documents are designed to support machine-generated “code stubs” to be used as scaffolding for server and client developers.

4.1 Essential Elements

The proposed ALPS format is rather small. It consists of five elements (`alps`, `doc`, `descriptor`, `ext`, and `link`) and nine attributes (`format`, `href`, `id`, `name`, `rel`, `rt`, `type`, `value`, and `version`). Describing the model in full is beyond the scope of this work. However, there is one important element in the ALPS format which is worth reviewing in order to get an idea of how the format differs from others mentioned in this paper: the `<descriptor>` element.

4.1.1 Descriptors

The most important element in ALPS is the `<descriptor>` element. It can occur in one of four "types":

- `semantic` : Describes data elements such as "firstname" or "hatsize"
- `safe` : Describes a safe transition such as an HTTP GET request
- `unsafe` : Describes a non-idempotent unsafe transition such as an HTTP POST request
- `idempotent` : Describes an idempotent, unsafe transition such as an HTTP PUT or DELETE request.

A typical element used to describe a "safe" transition (e.g. HTTP GET to a "home page") looks like this: `<descriptor id="home" type="safe">`

This is a guide for generating a message element in a common media type at runtime. For example, here is what it might look like in Collection+JSON:

`{"link" : {"rel" : "home", "href" : "..."}}`

Note that the `<descriptor>` element contains no `href` attribute. The URL is supplied by the server implementation and is of no interest when creating the service description.

Some descriptors may have parameters, too. Here is one with a single parameter and a possible HTML representation of the same descriptor.

```
<!-- ALPS -->
<descriptor id="search" type="safe">
  <descriptor id="fullname" type="semantic" />
</descriptor>

<!-- HTML -->
<form href="..." action="get">
  <input type="text" name="fullname" />
  <input type="submit" />
</form>
```

Descriptors can also dfine data elements that appear within a response. For example, below is a set of descriptors that describe state fields for a contact record (ALPS document authors can optionally nest `<descriptor>` elements).

```
<descriptor id="contact" type="semantic">
  <descriptor id="givenName" type="semantic" />
  <descriptor id="familyName" type="semantic" />
  <descriptor id="email" type="semantic" />
  <descriptor id="telephone" type="semantic" />
</descriptor>
```

4.1.2 Transitions, not Things or Functionality

While many of the service description formats focus on describing either functionality (WSDL, etc.) or things (Atom, etc.) ALPS describes individual transitions (`<descriptors>` of the type `safe`, `unsafe` and `idempotent`). These transitions may appear on any page. They may even all appear on the same page or they may appear on different pages over the life of the service. ALPS is not meant to tell implementors *where* transitions appear. Instead ALPS documents just indicate *what* transitions are possible and *how* they are completed (via parameters, protocol methods, etc.).

It is this focus on transitions over anything else that sets ALPS apart from the other formats covered here.

4.2 An ALPS Example

As an illustration, here is an example of an ALPS document that describes a contacts service that supports search. This shows how ALPS documents can be used to express actions and data elements within a simple problem domain.

```
<alps version="1.0">
  <!-- a hypermedia control returns contacts -->
  <descriptor id="search" type="safe" rt="contact">
    <descriptor id="name" type="semantic" />
  </descriptor>
  <!--  a contact: one or more may be returned -->
  <descriptor id="contact" type="semantic">
    <descriptor id="link" type="safe" />
    <descriptor id="givenName" type="semantic"
      href="http://schema.org/givenName"/>
    <descriptor id="familyName" type="semantic"
      href="http://schema.org/familyName" />
    <descriptor id="email" type="semantic"
      href="http://schema.org/email" />
    <descriptor id="telephone" type="semantic"
      href="http://schema.org/telephone"/>
  </descriptor>
</alps>
```

The above document identifies a transition block (`<descriptor id="search" type="safe" />`) and a data block (`<descriptor id="contact" type="semantic">`). The data block consists of four sub-elements (`givenName`, `familyName`, `email`, and `telephone`). In this description, activating the transition block will return one or more data blocks. Note that some data elements refer to content at the schema.org site[25]. ALPS documents are able to cite sources for individual descriptors in order to improve human understanding.

4.2.1 ALPS Implementations for Contacts

Since ALPS documents only describe services, the information they contain must be mapped to a target message and protocol at implementation time. Using the guide shown earlier, ALPS values for the `type` attribute (`semantic`, `safe`, `unsafe`, and `idempotent`) can be mapped onto protocol verbs and media type elements. Below is an elided HTML representation for the HTTP protocol.

```
<html>
  <head>
    <link rel="profile"
      href="http://example.org/contacts.alps" />
  </head>
  <body>
    <form href="..." action="get" class="search">
      <input type="text" name="name" value="" />
      <input type="submit" />
    </form>
    <ul>
      <li>
        <dl>
          <dt>givenName:</dt>
          <dd>...</dd>
          <dt>familyName:</dt>
          <dd>...</dd>
          <dt>email:</dt>
          <dd>...</dd>
        </dl>
      </li>
      ...
    </ul>
  </body>
</html>
```

Below is an elided Collection+JSON[6] representation for The Constrained Application Protocol (CoAP)[12].

```
{ "collection" :
  {
    "version" : "1.0",
    "href" : "...",
    "links" : [
      {"link" :
        {
          "rel" : "profile",
          "href" : "http://example.org/contacts.alps"
        }
      }
    ],
    "items" : [
      { "href" : "...",
        "data" : [
          {"name" : "givenName", "value" : "..."},
          {"name" : "familyName", "value" : "..."},
          {"name" : "email", "value" : "...",},
          {"name" : "telephone", "value" : "...",}
        ]
      },
      ...
    ],
    "queries" : [
      {"link" :
        {
          "rel" : "search",
          "href" : "...,
          "data" : [{"name" : "name", "value" : "..."}]
        }
      }
    ]
  }
}
```

4.2.2 Compliance for Clients and Servers

The reader may notice that the HTML/HTTP implementation does not render the `telephone` element in the data block. That implementation is, however, still considered ALPS compliant. The appearance of data or transition elements in an ALPS document is not a *requirement* for server implementors. Servers are free to select elements they wish to support. However, compliant client applications SHOULD to be prepared for all possible data and transition elements that are described in the ALPS document.

This is a reversal of common assumptions about client and server compliance on the Web. Most description formats describe things the server MUST implement and allow client applications to pick and chose the elements they wish to support. There are two important reasons for this reversal in the ALPS design.

First, ALPS documents do not *define* functionality (as do many service description formats). Instead ALPS documents *describe* a problem domain. ALPS documents provide a boundary around a problem by listing all the data and transitions that make up that domain. Just as Medical dictionary has a full set of words for that particular domain, physicians are not required to actually *use* them all with each patient the physician encounters.

Second, since ALPS documents describe a complete problem space, client applications are expected to support all the features described within the document in order to operate successfully with each server implementation it encounters. This enables the "serendipitous reuse" Charlton talks about[24].

4.3 Challenges for ALPS

The ALPS description format is still quite new and has not yet been used in any significant real-world implementations. However, based on a handful of test examples and experiments some challenges have been identified. These will need to be addressed if the ALPS format is to be widely used. Below are the most common challenges encountered to date when working with developers and service architects using the ALPS format.

This is is not exhaustive and may grow as more experiments and testing is done on the ALPS format.

4.3.1 No URLs

One of the most common challenges to people using the ALPS format is the lack of URLs within the description document. Most code generation tools expect to see URLs. Most human developers hand-writing client applications expect to see URLs, too. While removing URLs from the service document provides additional freedom for both client and server implementors, there are some developers who do not find this new freedom appealing.

Writing clients and servers from ALPS documents puts an additional burden on developers to focus on state machines and state transitions over URLs and objects. This is relatively new for client developers. This new focus is not easily supported by existing developer tooling which means developers must do this work by hand. Intially this adds extra work to the implementation process. It is possible new tooling will come along that supports this kind of description, but none seems forthcoming at this time.

4.3.2 Data Quality Descriptions

Some developers complain that ALPS does not easily support strong typing of response representations. This problem is similar to the "No URLs" problem. Most tooling is geared toward returning serialized objects, not state values expressed in hypermedia message models.

Server implementors often want the service description to include data quality information (e.g. "MUST be a string between 10 and 32 characters in length.". etc.). The current ALPS specifications do not support this since one implementation may have very different data quality rules than another. For example the `postalCode` data validation in Canada is quite different than the one used in the United States. Adding this to ALPS documents over-constrains it's use in other locales.

Client implementors would like to know the possible values for enumerated data elements (e.g. "List of valid shoe sizes", etc.). Like server-side quality checks, placing these within the service document limits it's use across similar (but varied) implementations and, so far, adding these elements to ALPS has been resisted.

4.3.3 Test-Driven Development Support

Another important challenge to the adoption of ALPS is it's use in implementations that support the Test-Driven Development (TDD)[30] approach. Essentially, clients and servers implementing an ALPS specification are likely to vary the location and even, in some cases, the responses to hypermedia requests. This complicates most TDD tooling that is geared for testing simple functions, parameters, and the expected return values. Developers fully committed to TDD and TDD tooling will be frustrated by the lack of tight coupling between the ALPS description and any given implementation.

4.4 Summary

The ALPS format (and the resulting implementations) often runs counter to a number of common programming practices. Most developer tooling (editors, test tools, etc.) do not work well with ALPS documents and/or the resulting applications that are ALPS-compliant. Some of this is due to a lack of useful tools. Some is due to programmer habits that may change over time as ALPS and ALPS-like service descriptions become more common. But until these challenges are addressed ALPS is not likely to be widely adopted and used.

5. CONCLUSION

There have been more service description formats created in the last two years than the previous ten. Most of them follow two main approaches: describe functionality or describe things. Both approaches lead to client applications tightly bound to a single instance of a service and unable to adapt over time. As new features and parameters are added, these clients ignore them or *hold their noses* and continue doing the same thing they've done since the first day they started running.

This paper looks at aspects of existing description formats that are problematic and suggests a new model for service description (and a sample implementation called ALPS) that makes it possible for client and server applications to share understanding without over-constraining implementations. These same characteristics make it possible for client applications to identify servers (via `profile` link relation values) that provide the same services and successfully interact with these servers even though these two parties never "met" before. It is through this shared understanding that clients can take advantage of "serendipitous reuse."

Finally, some important challenges to the adoption of ALPS-like were identified based on a small set of recent tests and experiments with developers using ALPS. Some challenges are related to human habit and others are related to available tooling. Most of these challenges will need to be resolved if ALPS-like service descriptions are to become common-place.

6. REFERENCES

[1] Nestor, J; Wulf, William Allan.; and Lamb, David Alex, "IDL, Interface Description Language" (1981). Computer Science Department. Paper 2412. http://repository.cmu.edu/compsci/2412

[2] WorldWideWeb: Proposal for a HyperText Project http://www.w3.org/Proposal.html

[3] Web Services Description Language (WSDL) 1.1, 2001 http://www.w3.org/TR/wsdl

[4] Connolly, (2002, October 23rd) "on media types for OWL (5.13)" http://lists.w3.org/Archives/Public/www-webont-wg/2002Oct/0162.html

[5] SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), 2007 http://www.w3.org/TR/soap12-part1/

[6] Collection+JSON - Hypermedia Type, 2011 http://amundsen.com/media-types/collection/

[7] Roy. T. Fielding, "REST APIs must be hypertext-driven", 2008 http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven#comment-742

[8] Web Application Description Language, 2009 http://www.w3.org/Submission/wadl/

[9] Swagger Core- API Declaration, 2013 http://github.com/wordnik/swagger-core/wiki/API-Declaration

[10] The Atom Publishing Protocol, 2007 http://www.ietf.org/rfc/rfc5023.txt

[11] XML Schema for WADL, 2009 http://www.w3.org/Submission/wadl/wadl.xsd

[12] Constrained Application Protocol (CoAP) - Internet Draft 18, 2013 http://tools.ietf.org/search/draft-ietf-core-coap-18

[13] RESTful Service Description Language (RSDL), 2012 http://g.mamund.com/szowd

[14] Home Documents for HTTP APIs - Internet Draft 03, 2012 http://tools.ietf.org/html/draft-nottingham-json-home-03

[15] XEP-0072: SOAP Over XMPP, 1999-2003 http://xmpp.org/extensions/xep-0072.html

[16] SMTP Transport Binding for SOAP 1.1, 2001 http://www.pocketsoap.com/specs/smtpbinding/

[17] SOAP over Java Message Service 1.0, 2012 http://www.w3.org/TR/SoapJms/

[18] Google APIs Discovery Service, 2012 http://developers.google.com/discovery/v1/reference/apis

[19] JSON Hyper-Schema: Hypertext definitions for JSON Schema - Internet Draft, 2013, http://json-schema.org/latest/json-schema-hypermedia.html

[20] RAML Version 0.8: RESTful API Modeling Language, 2013 http://raml.org/spec.html

[21] RESTdesc—A Functionality-Centered Approach to Semantic Service Description and Composition, 2012, http://g.mamund.com/javvz

[22] Verbough at el, "Efficient Runtime Service Discovery and Consumption with Hyperlinked RESTdesc", 2011 http://ruben.verborgh.org/publications/verborgh_nwesp_2011/

[23] A Brief History of the Development of SMDL and HyTime, 1994 http://www.sgmlsource.com/history/hthist.htm

[24] Stu Charlton, "Planned vs. Serendipitous Reuse", 2007 http://www.stucharlton.com/blog/archives/000165.html

[25] Schema.org, accessed January 2014 http://schema.org

[26] Extensible Messaging and Presence Protocol (XMPP): Core, 2011 http://tools.ietf.org/search/rfc6120

[27] Hypermedia-Oriented Design, 2011 http://g.mamund.com/qrkso

[28] Application-Level Profile Semantics (ALPS), accessed January 2014,
http://alps.io/spec/index.html

[29] The “profile” Link Relation Type - RFC6906, 2013
http://tools.ietf.org/html/rfc6906

[30] Kent Beck, Test-Driven Development: By Example. 2002, Addison-Wesley.

[31] HTML/Specifications (Accessed January 2013)
http://www.w3.org/community/webed/wiki/HTML/Specifications

[32] Voice Extensible Markup Language (VoiceXML) Version 2.0, 2004
http://www.w3.org/TR/voicexml20/

[33] Information Technology - Database Language SQL, 1992
http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt

[34] CSDL v3 Specification, 2009
http://msdn.microsoft.com/en-US/data/jj652004