Data Sharing for Cloud Computing Platforms

Marwan Sabbouh, Kenneth McCracken, Geoff Cooney
Social Platform
Here, a Nokia Business
Cambridge, MA, USA
Marwan.Sabbouh@here.com, Ken.McCracken@here.com, Geoff.Cooney@here.com

Abstract— **Cloud computing platforms consist of a set of reliable services that are run in the cloud. Typically, consumer applications use software development kits (SDKs) provided by the computing platform services to store, update, and retrieve instances of data in the cloud. Services provided by the cloud computing platform, expose different data modeling paradigms that consumer applications use to interact with the cloud. The service-specific data modeling paradigms and SDKs increase the complexity of data sharing between consumer applications that interact with the different services of the cloud computing platform. To make matters more complicated, it's not uncommon in an enterprise to find different groups using different cloud computing platforms. In this paper, we will describe a set of abstractions that can be used to abstract different computing platforms. The abstractions not only abstract the computing platform, but also enable the data discovery and sharing between applications. We will further show that these abstractions do not add substantial latency on the performance of the computing platform.**

Keywords- web protocols; data modeling language; NoSQL; Rest; API; JSON

I. INTRODUCTION

Most cloud computing providers offer a distributed datastore/database [1] [2] [3], in the form of a NoSQL engine or a NewSQL [4] [5] engine. These distributed databases expose a data modeling paradigm that their consumers can use to interact with the cloud system. For example, Amazon Web Services offers DynamoDB [6]; a fully managed NoSQL database service that makes it easy to store and retrieve data, for applications requiring varying levels of throughput. DynamoDB data modeling concepts include Tables, Items, and Attributes. Applications wishing to store their data in the cloud, can then define their tables, items and attributes as required by the distributed databases. However, what if multiple applications need to store and share their data. Certainly, the sharing of the data is enabled through the use of common data models [7] and common data protocols. Therefore, the use of a distributed database data modeling concepts, e.g. Tables, Attributes, and Items is typically not sufficient for the sharing of data.

The need for common data models requires the availability of the data modeling language as part of the data protocol. A cloud system data model not only contains the type definition, the listing of properties and their constraints, but also contains object-metadata specifying processing directives to the cloud system, property annotations, index definitions (if search queries are desirable for example), and access control policies.

In addition to the above requirements, there is also the requirement that the data model must be processed by the same components of the cloud system that process instance data. Hence, there is a need for the data modeling language to share a common object model with the rest of the instance data.

We believe that the above requirements are enabled through the use of a data interchange protocol providing:

1) A common object model

The object model stipulates that all data are instances of some type. The object model specifies a unique way to identify and type an object. The object model also specifies how to reference a remote object. In the context of a cloud system, the object model may also specify primary keys in the cloud.

2) A common serialization format

The serialization format, which in our case is JavaScript Object Notation (JSON) [8], eliminates the potential of having syntactic mismatches in the system. JSON was chosen for many reasons, particularly its simplicity, fast serialization/deserialization [9] as compared to Extensible Markup Language (XML), and the familiarity of many programmers with it.

3) A common exchange document

The reason for having a common exchange document is for applications to know a priori, the expected document structure. This in turn, eliminates the potential of any structural mismatches in the system.

4) A data modeling language

The data modeling language provides the necessary vocabulary for writing data models. Data models can serve a few purposes in the context of sharing data. First, they validate the instances. Second, they can be used by both consumers and providers of data to build a common understanding for a domain. Third, they contain access control policies and indexing directives.

5) Universal Resource Identifier (URI) conventions

URI conventions describe the address of the data in the cloud system. URI conventions make it possible to navigate

the exchange document to extract objects or instances, and a (property, value)-pair.

6) Representational state transfer (REST) APIs for storing and modifying the data

With URI conventions already in place, storing, creating, and retrieving data can easily be accomplished using REST [10] APIs. Once more, this API applies to both instance data and data models.

II. BACKGROUND

There are several solutions available, with each providing its own advantages and disadvantages. Google's Data Protocol (gData) [11] is a web protocol for reading, writing, and modifying data on the web. gData supports JSON serialization. The basic idea is that Google's internal services publish their data using gData, enabling consumers of the services to consume the data in a uniform way. The gData object model is that of an RSS/Atom feed. gData defines common definitions of certain objects, called Kinds in gData, but stops short of defining and using a data modeling language. Therefore, gData is an appropriate paradigm for publishing applications data, but may not be suited as a protocol for abstracting a NoSQL/NewSQL distributed database.

Open Data Protocol (OData) [12] is a web protocol that is backed by Microsoft. This protocol is used for creating and updating data using web technologies such as Atom feeds, JSON, and Hypertext Transfer Protocol (HTTP). OData provides an Entity Data Model (EDM) using Common Schema Definition Language (CSDL). CSDL is XML-based representation of the entity model exposed by an OData service. Additionally, OData provides a service model that allows the discovery of the capabilities of services. While OData claims that it "provides a uniform way to describe both the data and the data model", we note that in OData, there is a complete disconnect between the representation of the entity data model, which is in XML, and the representation of the instance data, which is an RSS/Atom feed. That is, the entity data model is not confined to the same object model as the services' data. Therefore, we viewed OData as a heavyweight protocol that offers much useful functionality, but we still required a simpler approach.

Protocol Buffers [13] is an approach for encoding, serializing, de-serializing, structured data in an efficient way. Protocol Buffers describes data structures in a textual format. Protocol Buffers tools are then able to consume the description of data structures and generate a proxy or stub [14] that handles the encoding, serializing, and de-serializing of structured data. While being very efficient on many levels, the use of Protocol Buffers requires the availability of those proxies and stubs at runtime for maximum efficiency. In turn, this makes it very difficult to introduce a new data model into production/live system in a seamless way. Furthermore, the data structure descriptions are not expressive enough lacking, for example, subclassing functionality.

We further considered the use of JSON Schema [15] alongside JSON to specify the data model. However, we quickly discovered that JSON Schema was not expressive enough to describe the indexing requirements and some of the access control policies that need to be specified. That is, while we were able to validate instances of data models using JSON Schema, we were unable to validate the data models themselves using JSON Schema, as the data models contained index definitions for search, and access control policies, whose support required disjoint property/object, among other features missing from JSON Schema. Since, JSON Schema and our data representation in JSON did not share a common object model; we were unable to process the data models using the same software components we had in place for the instance data. This meant that in an earlier implementation, the processing of data models was done manually.

Yet, another option that was considered but ultimately rejected was the use of a different modeling language that is expressive enough, e.g. Resource Description Framework (RDF) [16] and Web Ontology Language [17] (OWL). RDF/OWL is a powerful data modeling language designed for use on the web. It offers a powerful object model, as well as vocabularies for defining types, properties, annotations, restrictions, and many other useful data modeling primitives. However, the issues with using RDF/OWL are the lack of JSON serialization for RDF/OWL, and the open world assumption of RDF/OWL. The open world assumption states that any statement that is not known to be true is not necessarily false. These issues present significant hurdles in the adoption of RDF/OWL in cloud-based systems that require validation of data. Further, that choice would have presented latency challenges for processing the instance data fast enough.

Hence, we have arrived at the conclusion that the Data Interchange Protocol is needed.

III. PROTOCOL SPECIFICATION

Data Interchange Protocol (DIP) defines an object type system using JSON. DIP uses special attributes, e.g. _type and _name, for ALL data, making all DIP Objects "typed", and "named". DIP uses a special attribute _id to specify the key of the object in the distributed database. DIP provides standard DIP object types, e.g. Entry and Field for the creation of types, and for associating properties with a type. DIP defines URI conventions for addressing DIP objects and properties. DIP data modeling language defines an inheritance model allowing Is-A relationships between types. Additionally, DIP supports data modeling features such as disjoint properties, property restrictions, property intersections (implications), property cardinality constraints, disjoint objects, and intersection objects, making the DIP modeling language expressive enough to validate both DIP object instances and DIP data models themselves. DIP data models can contain more than object type definitions. Specifically, they can also include access control policies, indexing policies, and other meta-models concerning the type being defined. While out of scope for this paper, we simply note that validating a DIP data model entails more work than simply validating the type being defined. For example, the data model validator need to ensure that valid

access controls settings are applied on the type being defined.

A. DIP Object Model

DIP object model treats every object as instance of some type. DIP object model defines a uniform way for 1) creating types that are unique in the cloud system; 2) instantiating instances of those types; 3) identifying those instances in the cloud; 4) referencing remote instances; 4) addressing the instances and their properties. DIP objects are exchanged using DIP documents. A DIP document is comprised of a JSON list with each member of the list is a JSON object. Fig. 1, shows an example.

```
[{
"_type": "Entry",
"_id": "Field",
"_name": "Field",
"_comment": "base class",
“extends”:null
…},{…}]
```

Fig. 1. DIP document

DIP defines its object model using system properties. DIP object model reserves the use of all property names beginning with _. We refer to property names beginning with _ as System Properties. System Properties have special meanings, and they can occur in any object. They are integral to the system and they are not defined in any data model. DIP has the following System Properties that relate to the object model: _type, _name, _id, _comment, _ref, _uri.

_type specifies the type of a DIP object, meaning that all property (name, value)-pairs in the object that are non-System Properties are associated with the type specified as the value of _type. The value of _type is a single value string. Fig. 1 shows that the value of _type is "Entry". The value of _type is interpreted as a relative URI (see _uri description). In DIP, Entry is used to define new types. DIP objects are required to specify _type.

_name specifies the name of a DIP object. Since _name is Field, Fig. 1 shows that the type being defined is Field. The value of _name is a single value string, and is unique in the context of a DIP document. Hence, Fig. 2 shows a DIP document containing a single logical object comprised of two objects with the same name. Objects are required to specify _name.

```
[{
"_type": "Entry", "_id": "Field", "_name":
"Field",
"_comment": "base class", …
}, {"_type": "Entry", "_name": "Field",
"extends": null}]
```

Fig. 2. Single logical object in DIP

In the DIP object model, Fig. 2 and Fig. 1 are semantically equivalent as they define a single logical object named Field with its "extends" property set to null.

_id when combined with _name is typically used to specify the key in a NoSQL database that holds the DIP document as its value. The value of _id is a single value string, which represents a unique identifier in the cloud system. _id must be present in the first object of a DIP document. When "_id" is not present in DIP objects, its value is assumed to be that of the first object.

_uri is used to combine _id, _name, _type in a single property (name, value)-pair. _uri is used as an alternative notation for _id, _name, _type. Therefore, Fig. 1 could be stated as shown in Fig. 3.

```
[{
"_uri": "/Field/Field:Entry",
"_comment": "base class",
"extends": null
…
},
{…}]
```

Fig. 3. DIP document using _uri

Fig. 3 shows the same logical object as Fig. 1 and Fig. 2. The value of "_uri" is a single value string and has the form /_id/_name:_type. Hence, Fig. 3 shows that _id is Field, _name is Field, and _type is Entry. Fig. 3 also shows the value of "_uri" to be a relative URI which can be expanded to be an absolute URI by the system.

_comment provides a textual description of the object being defined.

_ref provides a mechanism to referring to remote objects. The value of _ref is either a string or a list of string. This string has the same form as the value of_uri. However, the difference between _ref and _uri is that _ref points to an object that was defined elsewhere, while _uri defines the object. Fig. 4 shows a typical use of _ref.

```
[{
"_uri": "/C544C14C-51F0-0001-5FB6-
262014061F9F/Marwan:Person",
"_comment": "instance of Person named
Marwan",
"employer": {"_ref": "/C544d14d-51F0-0001-
5FB6-2620134344F9F/Nokia:Company"}
…
}]
```

Fig. 4. Typical use of_ref

Fig. 4 shows an instance Marwan of type Person. This instance has a property employer. Its value is a reference to an instance Nokia of type Company. It is defined in DIP document C544d14d-51F0-0001-5FB6-2620134344F9F.

B. Modeling of DIP data

DIP Data can be modeled as a property graph [18]. A DIP object is comprised of nodes and edges. A node is characterized by either the value of _uri, or the combined value of _name, and _id. Typically, such node has outgoing edges from it. The properties that are part of the DIP object are either attributes of a node, or outgoing edges from a node. When the property value is another DIP object containing System Property _ref, the value of _ref characterizes a node with incoming edge. That is, when the value of the property is interpreted as a URI, it is a node in the graph. Fig. 5, shows the representation of Fig. 4 as a property graph.

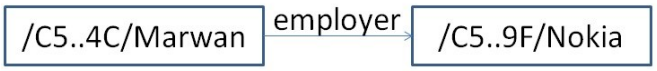

Fig. 5. Graph representation of Fig. 4

C. DIP Type System

The DIP type system supports the basic types and structures defined by JSON, e.g. array, object, number, string, true, false, null. That is, a DIP object is a valid JSON object. Entry is the type that all other types use. Entry is used to define new types. The type Field is used to define properties and to associate them with a type. To specify that the DIP object is an instance of type X, it suffices to set the _type property to X in the DIP object.

1) Type Entry

Entry is the base type in DIP type system. Fig. 6 shows the formal description of Entry. The first object in the DIP document specifies Entry to be a type. The second object in the DIP document is an instance of type Field. This instance contains the usual System Properties, indicating that the instance name is extends. In addition to the System Properties, it contains other properties: dataType, isFieldOf, and sequence. In summary, Fig. 6 says that Entry is a type, it has a property called extends. The value of extends is a list of string. The sequence number of extends is 1. Furthermore, extends is a field of Entry. DIP adopts the convention of grouping all properties belonging to type in one DIP document.

Note that, in order for the Entry document to specify an instance of Field, Field must have been defined as a type in another document. The definition of Field is described next.

```
[ {
      "_id": "Entry", "_type": "Entry",
      "_name": "Entry",
      "_comment": "base class"
   },{
      "_type": "Field",   "_name": "extends",
      "dataType": [ "string"] ,
      "isFieldOf": "Entry",  "sequence": 1
   }]
```

Fig. 6. Entry definition

2) Type Field

Fig. 7 shows excerpts from the definition of Field.

```
[{
      "_id": "Field", "_type": "Entry",
       "_name": "Field",
   },{
      "_type": "Field","_name": "dataType",
      "dataType": "any", "minCardinality": 1,
      "isFieldOf": "Field",  "sequence": 1
   },{
      "_type": "Field","_name": "isFieldOf",
      "dataType": "string",
      "isFieldOf": "Field","sequence": 2
   },… {
      "_type": "Field","_name": "sequence",
      "dataType": "int",
"isFieldOf": "Field","sequence": 13
   },…]
```

Fig. 7 Field definition

The first object in the Field document states that Field is a type. The remaining three objects in the Field document define isFieldOf, dataType, and sequence as instances of Field. That is, these instances can occur in any object of type Field. There are other properties defined for type Field that are not shown here. Notable among them are dataType, minCardinality, maxCardinality, intersectWith, and disjointWith. We offer a brief description of each of those properties.

When a user creates a type T in DIP, and defines property p for that type:

if p has its dataType attribute set to "string" or "int" or "Boolean", or "number", this means every instance of T must have the property's value a scalar corresponding with the type. Otherwise, the instance is not valid.

if p has its dataType attribute set to a (relative) URI, e.g. "/:Address", this means the value of p is an instance of the type specified in the URI, e.g. Address. Fig. 6 shows an

example of how to specify a property, i.e. extends, whose value is a List of string.

if p has its minCardinality attribute set to a number v, this means every instance of T must contain at least v occurrences of that property. Otherwise, the instance is not valid.

if p has its maxCardinality attribute set to a number v, this means every instance of T must contain at most v occurrences of that property. Otherwise, the instance is not valid.

if p has its intersectWith set to another property p2, this means every instance of T with p set to a value must also have p2 set to a value. Otherwise, the instance is not valid.

if p has its disjointWith set to another property p2, this means every instance of T with p set to a value must not have p2. Otherwise, the instance is not valid.

3) Multi-typing an Instance

DIP has the capacity to express that an object is of type A, and of type B. At first look, one might think that DIP does not allow multi-typing of instances due to the fact that the value of _type is a single value string. However, DIP accomplishes this feat by enclosing two different objects having the same value for _name, and different values for _type in the same DIP document. This is shown in Fig. 8.

```
[{
      "_id": "C5254000-….BD20F34E142C",
      "_type": "Person",
      "_name": "Marwan",
       "gender": "male"
   },{
      "_type": "Worker",
      "_name": "Marwan",
       "employer": "Nokia"
   }]
```

Fig. 8. Instance multi-typing

Fig. 8 shows an instance named Marwan having two types: Person, and Worker. This approach to multi-typing an instance keeps the properties of each type separated from each other. For example, Fig. 8 shows that employer is a property belonging to Worker, by the virtue of including this property in the object with _type set to Worker. Other approaches to achieve multi-typing are almost guaranteed to be more complicated, as they would need to introduce prefixes for each type that help distinguish the properties belonging to one type from the properties belonging to another type.

4) Type Object

In addition to specifying constraints on properties, DIP permits the specifying of intersection, and disjoint constraints on objects. Therefore, the DIP type Object defines the properties intersectWith , and disjointWith whose values must be interpreted as URIs. I will elaborate briefly on those properties without showing the definitions for the sake of brevity.

When a user defines a type T to be an instance of Entry and of Object;

If T sets intersectWith to type T2, then, any instance I of T, must also be an instance of T2. Otherwise, the DIP document is not valid.

If T sets disjointWith to type T2, then, any instance I of T, must not be an instance of T2. Otherwise, the DIP document is not valid.

Fig. 9 shows a typical use of type Object. Fig. 9 shows that the DIP document, if it contains an index, i.e. an instance of SCBEIndex, the instance of index must also be an instance of scbeAccessControl. That is, if the document defines an index, it must also define access control on the index.

```
[{
    "_type": " Object",
    "_name": " SCBEIndex"
    "intersectWith": ["scbeAccessControl"]
  },{
    "_type": " Entry",
    "_name": " SCBEIndex"
  }]
```

Fig. 9. Example use of type Object

5) Extension Mechanism

DIP supports a mechanism to express is-a relationships between objects. The processing of object A extends object B is as follows:

- Any instance of type A must also be an instance of type B. This is accomplished using the multi-typing serialization described in section C.3.
- Therefore, any document that contains an instance of type A, that same document must also contain the serialization of that instance of type B.
- All property and object constraints defined on type B are applied on the instance of type B.
- All property and object constraints defined on type A are applied to the instance of type A.

Note that the definition of extends is used in Fig. 6.

6) Serializing Properties with Cardinality Greater Than One

DIP serialization offers a straightforward approach to expressing multiple occurrence of a property. Fig. 10 shows an example.

```
[{
      "_id": "Person",
      "_type": "Entry",
      "_name": "Person"
   }, {
       "_type": "Field",
      "_name": "address",
      "dataType": "string",
      "isFieldOf": "Person",
      "sequence": 1
   },   {
      "_type": "Field",
      "_name": "address ",
      "isFieldOf": "Person",
      "dataType": "./:Address"
}]
```

Fig. 10. Definition of property address

Fig. 10 shows that the property dataType occurs twice to state that the property address can have a value either of type string or of type Address.

7) Property Restrictions

Earlier in the paper, we have seen examples on how to specify an object property, or property whose value is an instance of a certain type. Sometimes, it is useful to further restrict the value of an object property to not only state that its values are instances of a certain type, but also to specify a restriction on the property belonging to that type. DIP makes that possible by manipulating the format used for _uri. Recall that the format of _uri is of the form "/ _id/_name:_type" . To specify a restriction on the value of _type, we simply manipulate the value of _uri. These are few examples:

- To specify any instance of type Field, we write the following: "./:Field"
- To specify any instance of type Field in DIP document c54…789, we write the following: "c54…789/: Field"
- To specify an instance of type Field by name, we write the following: "./name: Field"
- To specify any instance of type Field, but with the added restriction of having dataType to be string, we write the following: "./:Field; dataType = string"
- To specify any instance of type Field, but with the added restriction of having dataType to be string *and* isFieldOf to be systemProperties, we write the following: "./:Field; _op= and; isFieldOf = systemProperties; dataType = string"
- To specify any instance of type Field, but with the added restriction of having dataType to be string *or* isFieldOf to be systemProperties, we write the following: "./:Field; _op= or; isFieldOf = systemProperties; dataType = string"

The above notations for property restrictions have several applications particularly in data modeling languages. For example, DIP data modeling language wishes to provide subclassing functionalities based on restriction of properties. Fig. 11, shows how this can be done using DIP.

```
[{
      "_id": "stringField",
      "_type": "Entry",
      "_name": "string Field",
      "_comment": "Field description",
      "extends": ["./:Field;dataType=string"],
   }]
```

Fig.11. Example use of property restriction

Fig. 11 shows the definition of type stringField which is the class of all instances of Field, with dataType = string. Suppose the existence of an instance of Field, but with datatype = number, that instance would not be a member of the type stringField.

8) Indexing of DIP Documents

It is common for cloud systems to offer indexing and search solutions. In this case, applications' data stored in the cloud are indexed in order to provide search functionality. To implement this functionality, cloud systems often allow applications to define their own index definition file. In this approach, each application would have its own index definition. DIP object model makes it possible to index all applications' data using a single index definition. The table header below shows a typical index definition for any DIP document. Other variation of this header which includes other columns is also possible provided that it maintains the columns shown in Table I.

Table I Index definitions for DIP documents

id (key, string)	_type (key, string)	_name (key, string)	property (key, string)	value (multivalue ,string)	_ref (string)	simpleType

Table II shows the necessary indexed fields that comprise an index definition. The fields tagged as key form a compound key indicating a unique row in the table. The value field is tagged multivalued indicating that field's value is a list. The column labeled simpleType contains the type of the value when it is a simple type, e.g string, int, etc. The column labeled _ref is an indicator that the value is a pointer

to another object. Table II partially shows the index data for Fig. 10.

Table II Index data of Fig. 10

Person	Entry	address	dataType	["string"]		str
Person	Entry	address	dataType	[" Address"]		str

D. REST API for Create, Read, Update, and Delete Operations

Web data protocols offer their consumers a REST API to create, read, update, and delete (CRUD) their data. DIP offers a fully functional REST API. In this paper, we provide a short summary of these operations in Table III.

From a conceptual perspective, DIP permits clients to create/get/delete a DIP document, and to update/insert/delete any object in the DIP document. In addition to working at the document and object granularities, DIP permits clients to get/update/insert/delete any properties in the DIP document. A few additional observations to make on the API:

- The HTTP request body for all HTTP Put/Post operations is a DIP document. Also, the HTTP response body for all HTTP Get, is also a DIP document. Therefore, whether retrieving a single object or the entire document, the structure of the reply is always the same.
- A HTTP Get request on URI /_id/_name:T would also resolve to URI /_id/_name:T_1 if T_1extends T, and so on. In that case, the returned DIP document contains the union of the operations on the two URIs.
- Upon receiving a HTTP Post/Put request on URI /_id/_name:T_1, API enforces that the instance is also of type T if T_1 extends T.
- Upon receiving a HTTP delete request on URI /_id/_name:T , API also executes delete operation on URI /_id/_name:T_1 if T_1 extends T.

It is an error to create the same DIP document more than once. That is, once the document is created it can only be modified or deleted in subsequent operations.

Table III DIP operations using Rest

URL convention	HTTP verb	description
/_id	Post/Delete / Get	Create/Remove/Get a DIP document
/_id/_name	Get/Post/ Delete	Retrieve/update/Delete the named instance
/_id/_name:type	Get/Post/ Delete	Retrieve/update/Delete the named/typed instance

IV. PERFORMANCE EVALUATION

For DIP to be successful, it must not add significantly to the latency of the NoSQL database. The majority of the latency introduced by DIP comes from the validation of DIP instances. The instance validation is comprised of 1) validate property disjointness ; 2) validate property intersection; 3) check property data types; 4) check for required properties; and 5) check property values and enumeration constraints. It is important to know that this validation time is a function of the number of properties in the object, and not the size of the instance. The tests were run on a personal computer running in Windows 7, with CPU I7-2640M, and 8GB of memory. Most of the overhead imposed by DIP takes place on update operations and not on get. Of that overhead, most of it is spent on validation. We observed that the validation time represents about 6% of our average update operation latency for payloads with less than 200 properties. For large sized payloads, e.g. the number of properties approaching 1000, the validation time becomes more significant. Therefore, these numbers clearly demonstrates that validation can happen synchronously, e.g. as part of the request workflow, for smaller payload size. However, for larger payload size, validation should take place asynchronously after a request has been acknowledged as successful by the server, e.g. by having the server responds to the sender with HTTP status code 202, and then continue with the validation. Furthermore, we acknowledge that these numbers can further be improved by utilizing more efficient algorithms for validation. This can be the subject of another paper.

V. DISCUSSION

In this section, we describe our experience using the protocol. There are two internal projects that successfully used this protocol. In one of the project, we used DIP to express constraints related to cloud-based social platform. We found DIP to be very expressive, allowing us to state things like "all indices must define access control", "this mode of access control on this index excludes this other mode of access control on instance data", and "you can only define an index on this property if it has been defined in the data model". Particularly useful was the use of meta-modeling in DIP enabled by the multi-typing of instances. Also in the same endeavor, disjoint property, disjoint object and intersection property and intersection object proved very useful.

Related to the stability of DIP as an abstraction layer for distributed databases, DIP once more showed its worth as we were able to migrate from our own instantiation of Voldemort (Sumbaly, et al., 2012) to DynamoDB on AWS without any impact on client applications.

Regarding the use of the URI convention, we received feedback that it would be best if the type of the object is represented as a matrix parameter. Representing the type in the URI is the result of identifying each object in the system by id and type. As we improved our system to identify object solely by ID, we kept the type in the path so as not to break backward compatibility.

Since in cloud system, the same data tend to live in distributed databases and on other clusters for further analysis by recommendation engines, it is therefore important that we are able to treat the DIP data as a graph. Note that as the data is first stored in the distributed database, instance validation plays a key role to ensure the quality of the data. However, as data is analyzed by recommendation engines, open world assumptions take precedence as we are interested in deriving knowledge from existing one. To this end, we translated DIP to Resource Description Framework (RDF) fairly easily. This is an important point as an earlier attempt for us to use RDF as the entry point into the cloud was met by strong resistance from developers who are unaware of RDF's inference rules.

Related to the use of the document structure as a list of objects, we also found that to be very convenient as it enabled an application to embed two different objects in the same document, therefore guaranteeing atomic transactions for a single key, even in the presence of a distributed database that does not offer this functionality. However, a limitation to this approach is the payload size at about 60 KB.

VI. Conclusion

In this paper, we presented the data interchange protocol for sharing data. As part of that, we presented the data interchange protocol object model, typing system, uniform indexing of data, and graph modeling of the DIP data. We also reported our experience with the protocol. Our next step is to submit this as a draft standard to one of the standard bodies.

Acknowledgment

This work has benefited from the many attempts of doing such a system internal to Nokia, starting with the work on the Unified API and culminating in the development of the Social Platform.

References

[1] Fay, D. Jeffrey et. al., "Bigtable: a distributed storage system for structured data," in 7th symposium on Operating systems design and implementation, Seattle, November, 2006.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, "Dynamo: amazon's highly available key-value store," in Twenty-first ACM SIGOPS symposium on Operating systems principles, 2007.

[3] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, A. L. Y. Li and V. Yushprakh, "Megastore: Providing Scalable, Highly Available Storage for Interactive Services," in Conference on Innovative Data system Research (CIDR), pp. 223-234, 2011.

[4] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem and P. Helland, "The end of an architectural era: (it's time for a complete rewrite.," in 33rd international conference on Very large databases, p. 1150–1160, 2007.

[5] J. Starkey, "Database management system". USA Patent 8224860, 17 07 2012.

[6] "Amazon DynamoDB," Amazon, [Online]. Available: http://aws.amazon.com/dynamodb/. [Accessed 23rd July 2013].

[7] D. Gagne, M. Sabbouh, S. R. Bennett and S. Powers, "Using Data Semantics to Enable Automatic Composition of Web Services.," in IEEE International Conference on Services Computing (SCC 06),Pg. 438-444, Chicago USA, 2006.

[8] "The application/json Media Type for JavaScript Object Notation (JSON)," IETF RFC 4627, 2006.

[9] "jvm-serializers Wiki," GitHub, [Online]. Available: https://github.com/eishay/jvm-serializers/wiki. [Accessed 23rd July 2013].

[10] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," University of California Irvine-PhD dissertation, Irvine, 2000.

[11] "Google Data Protocol," Google, [Online]. Available: https://developers.google.com/gdata/. [Accessed 23 July 2013].

[12] "Open Data Protocol," Microsoft, [Online]. Available: http://www.odata.org/introduction/. [Accessed 23 July 2013].

[13] "Protocol Buffers," Google, [Online]. Available: https://developers.google.com/protocol-buffers/. [Accessed 23 July 2013].

[14] M. Shapiro, "Structure and Encapsulation in Distributed Systems: the Proxy Principle," in 6th international conference on Distributed Computing Systems, Cambridge, Ma, 1986.

[15] "A JSON Media Type for Describing the Structure and Meaning of JSON," IETF Internet Draft , 2011.

[16] "Resource Description Framework," W3C Recommendation, Cambridge, 2004.

[17] "Web Ontology Language," W3C Recommendation, Cambridge, 2004.

[18] M. Rodriguez, "Knowledge Representation and Reasoning with Graph Databases," [Online]. Available: http://markorodriguez.com/2011/02/23/knowledge-representation-and-reasoning-with-graph-databases/. [Accessed 23 July 2013].

[19] "project Voldemort," [Online]. Available: http://www.project-voldemort.com/voldemort/. [Accessed 24 July 2013]..