Internet Engineering
Task Force
Internet-Draft
Intended status: Informational
Expires: August 8, 2009

A. Brashears
M. Hamrick, Ed.
M. Lentczner
Linden Research, Inc.

February 4, 2009

Linden Lab Structured Data
draft-hamrick-llsd-00

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on August 8, 2009.

Copyright Notice

Abstract

This document describes the Linden Lab Structured Data (LLSD)

abstract type system, interface description and serialization formats. LLSD is a language-neutral facility for maintaining and transporting structured data. It provides dynamic data features for loosely-coupled collections of software components, even in statically-typed languages. LLSD includes an abstract type system, an interface description language (LLIDL) and three canonical serialization schemes (XML, JSON and Binary).

Table of Contents

1. Introduction 4
1.1. Requirements Language 4
2. Abstract Type System 4
2.1. Simple Types 5
2.1.1. Undefined 5
2.1.2. Boolean 5
2.1.3. Integer 6
2.1.4. Real 6
2.1.5. String 6
2.1.6. UUID (Universally Unique ID) 7
2.1.7. Date 8
2.1.8. URI (Uniform Resource Identifier) 8
2.1.9. Binary 8
2.2. Composite Types 8
2.2.1. Array 8
2.2.2. Map 9
2.3. Converting Between Real and String Types 9
2.4. Converting Between Date and String Types 9
3. Serialization 10
3.1. XML Serialization 10
3.1.1. Serializing Simple Types 10
3.1.2. Serializing Composite Types 11
3.1.3. Example of XML LLSD Serialization 11
3.2. JSON Serialization 12
3.2.1. Examples of JSON LLSD Serialization 13
3.3. Binary Serialization 13
3.3.1. Example of BINARY LLSD Serialization 15
4. Interface Description Language 18
4.1. Abstract Data Types and Names 18
4.2. Abstract Data Structures 18
4.3. Variant Data Structures 19
4.4. Variant Discriminators 20
4.5. Resource Description 20
5. IANA Considerations 20
6. MIME Type Registrations 21
6.1. MIME Type Registration for application/llsd+xml 21
6.2. MIME Type Registration for application/llsd+json 22

6.3. MIME Type Registration for application/llsd+binary 23
7. Security Considerations 25
8. References 25
8.1. Normative References 25
8.2. Informative References 26
Appendix A. ABNF of Real Values 27
Appendix B. XML Serialization DTD 28
Appendix C. ABNF of LLIDL 28
Authors' Addresses 30

1. Introduction

Linden Lab Structured Data (LLSD) is an abstract type system intended to provide a language-neutral facility for the representation of structured data. It provides a type system, a serialization system and an interface description language.

The type system of LLSD defines nine simple types (Undefined, Boolean, Integer, Real, String, UUID, Date, URI and Binary) and two composite types (Array and Map.) It is used to represent an ideal dynamic type system in programming languages that may not exhibit dynamic type behaviors. This type system is advantageous in computing environments that make use of loosely-coupled components, each of which may be implemented in a different programming language.

When loosely-coupled systems need to communicate structured data, LLSD instances are serialized into a neutral format for transmission across a process or system boundary. LLSD instances may be serialized into one of three defined formats: XML, JSON and binary.

When meta-information regarding LLSD instances is required, an interface description language (LLIDL) may be used to define the structure of LLSD instances. LLIDL is especially suited to describing the structure of requests and responses in distributed systems using representational state transfer (RESTful) semantics.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Abstract Type System

The abstract type system describes the semantics of LLSD data passed between two systems. These types characterize the data when serialized for transport, when stored in memory, and when accessed by applications.

The types are designed to be common enough that native types in existing serializations and programming languages will be usable directly. It is anticpiated that LLSD data may be serialized in systems with fewer types or stored in native programming language structures with less precise types, and still interoperate in a predictable, reliable manner. To support this, conversions are defined to govern how data received or stored as one type may be read as another.

For example: If an application expects to read an LLSD value as an Integer, but the serialization used to transport the value only supported Reals, then a conversion governs how the application will see the transported value. Another case would be where an application wants to read an LLSD value as a URL, but the programing language only supports String as a data type. Again, there is a defined conversion for this case.

The intention is that applications will interact with LLSD data via interfaces in terms of these types, even if the underlying language or transports do not directly support them, while retaining as much direct compatibility with those native types as possible.

An LLSD value is either a simple datum or a composite structure. A simple data value can have one of nine simple types: Undefined, Boolean, Integer, Real, String, UUID, Date, URI or Binary. Composite structures can be either of the types Array or Map.

2.1. Simple Types

For each type, conversions are defined to that type. That is, if a process is accessing a particular LLSD value, and treating it as a particular type, but the underlying type (as transmitted, or stored in memory) is different, then the indicated conversion, if defined, is applied. If a conversion is not specified from a particular type, then if a value of that type is accessed, the result is the default value for the expected type. For example: When reading a value as an integer, if the underlying value is binary, then the value read is zero.

2.1.1. Undefined

Data of type Undefined has only one value, called undef. The default value is undef. There are no defined conversions to Undefined.

The Undefined type is a placeholder for a value.

2.1.2. Boolean

Data of type Boolean can have one of only two values: true or false. The default value is false.

Conversions:

Integer A zero value (0) is converted to false. All other values are converted to true.

Real A zero value (0.0) and invalid floating point values (NaNs) are converted to false. All other values are converted to true.

String An empty String is converted to false. Anything else is converted to true.

2.1.3. Integer

Data of type Integer can have the values of natural numbers between -2147483648 and 2147483647 inclusive. The default value for Integer is zero (0).

Conversions:

Boolean The value true is converted to the Integer 1. The value false is converted to the Integer 0.

Real Real are rounded to the nearest representable Integer, with ties being rounded to the nearest even number. Invalid floating point values (NaNs) are converted to the Integer 0.

String The string is first converted to type Real, see Section 2.3. Then the resulting Real is converted to Integer as specified above.

2.1.4. Real

Data of type contain signed floating precision numeric values from the range available with IEEE 754-1985 64-bit double precision values, as well as the special non-numeric values (NaNs and Infs) available with that format. The default value for Real is zero (0.0).

Conversions:

Boolean The value true is converted to the floating point value 1.0. The value false is converted to the floating point value 0.0.

Integer Integers promoted to floating point values are converted to the nearest representable number.

String See Section 2.3.

2.1.5. String

Data of type String contain a sequence of zero or more Unicode code points. The default value for String is a sequence of zero code points, the empty string ("").

The characters are restricted to the following code points:

U+0009, U+000A, U+000D

U+0020 through U+D7FF

U+E000 through U+FFFD

U+10000 through U+10FFFF

Strings may be normalized during transport, storage or processing. When an implementation does normalize, it should use Normalization Form C (NFC) described in Unicode Standard Annex #15 [TR15]. Line endings may be normalized to U+000A.

Conversions:

Boolean The value true is represented as the string "true". The value false is represented as the empty string ("").

Integer Integers converted to Strings are represented as signed decimal representation.

Real See Section 2.3.

UUID UUIDs converted to Strings are represented in the 36 character, 8-4-4-4-12 format defined in RFC 4122 [RFC4122].

Date See Section 2.4.

URI URIs converted to Strings are simply Unicode representations of the URI.

2.1.6. UUID (Universally Unique ID)

UUIDs represent a universally unique identifier. Data of type UUID is a 128 bit identifier with a structure defined in RFC 4122 [RFC4122]. The default UUID value is the null UUID, (00000000-0000-0000-0000-000000000000).

Conversions:

String A valid 8-4-4-4-12 string representation of a UUID is converted to the UUID it represents. All other values are converted to the null UUID (00000000-0000-0000-0000-000000000000).

2.1.7. Date

Dates represent a moment in time. Data of type Date may have the value of any time in the from January 1, 1970 though at least January 1, 2038, to at least second accuracy. The default date is defined as the beginning of the Unix(tm) epoch, midnight, January 1, 1970 in the UTC time zone.

Conversions:

String See Section 2.4.

2.1.8. URI (Uniform Resource Identifier)

Data of type URI has the value of a Uniform Resource Identifier as defined in RFC 3986 [RFC3986]. The default URI is an empty URI

Conversions:

String The characters of the String data are interpreted as a URI, if legal. Other Strings results in the default URI.

2.1.9. Binary

Data of type Binary contains a sequence of zero or more octets. The default Binary is a sequence of zero octets.

There are no defined conversions for Binary.

2.2. Composite Types

LLSD values can be composed of other LLSD values in two ways: Arrays or Maps. In either case, the values with the composite can be any heterogeneous mix of other LLSD types, both simple and composite.

2.2.1. Array

An Array is an ordered collection of zero or more values. The values are considered consecutive, with no gaps. The value undef (of type Undefined) may be used to indicate, within an Array, an intentionally left out value.

Arrays are considered to have a definite length, including any leading or trailing undef values in the sequence. This length can be viewed by an application. Accessing beyond the end of an array acts as if the value undef were stored at the accessed location. Nonetheless, systems that transmit or store Arrays SHOULD NOT add or remove undef values at the end of an Array value, so as to make a

best effort to retain the definite length as originally created.

2.2.2. Map

A Map is an unordered collection of associations between keys and values. Within a given Map value, each key must be unique, each with one value. Keys are String values. The associated values can be of any LLSD type.

Maps are considered to have a definite set of keys, including keys whose associated value is undef. The number of such keys, and set of keys can be accessed by an application. Accessing a value for a key that is not in a Map value's key set acts as if the value under were stored at that key. Nontheless, systems that transmite or store Maps SHOULD NOT add or remove keys associated with undef to a Map value, so as to make a best effort to retain the key set as originally coreated.

Note on key equality: Two keys are considered equal if they contain the same number and sequence of Unicode codepoints. Since keys are String values, and String values may be normalized on transport or storage, it follows that only String values that are already normalized as allowed by the String type are reliable as Map keys. Since the Maps are intended to be primarily used with keys set forth in protocol descriptions, this not a particular problem. However, if arbitray user supplied data is to be used as key values in some application, then the possibility of normalization and perhaps key collision during transport must be considered.

2.3. Converting Between Real and String Types

Real values are represented using the ABNF provided in Appendix A

2.4. Converting Between Date and String Types

The textual representation of Date values is based on ISO 8601 [ISO8601], and further specified in RFC 3339 [RFC3339]. When Date values are converted to or from String values, the character sequence of the string must conform to the following production based on the ABNF in RFC 3339 [RFC3339]:
full-date "T" partial-time "Z"
When converting from String values, if the sequence of characters does not exactly match this production, then the result is the default Date value.

3. Serialization

When used as part of a protocol, LLSD is serialized into a common form. Three serialization schemes are currently defined: XML, JSON and Binary.

3.1. XML Serialization

XML serialization of LLSD data is in common use in protocols implementing virtual worlds. When used to communicate protocol data with a transport that requires the use of a Type, the type 'application/llsd+xml' is used.

When serializing an instance of LLSD structured data into an XML document, the DTD given in Appendix B is used. This DTD defines elements for each of the defined LLSD types. Immediately subordinate to the root LLSD element, XML documents representing LLSD serialized data include either a single instance of an simple type (Undefined, Boolean, Integer, Real, UUID, String, Date, URI or Binary) or a single composite type (Array or Map).

3.1.1. Serializing Simple Types

Most simple types are serialized by placing the string representation of the data between beginning and ending tags associated with the value's type. This is true for undefined, boolean, integer, real, UUID, string, date and URI typed values. Values of type binary are serialized by placing the BASE64 encoding (defined in RFC 4648 [RFC4648]) of the binary data within beginning and ending 'binary' tags. It is expected that future versions of this specification may allow encodings other than BASE64, so the mandatory attribute 'encoding' is used to identify the method used to encode the binary data.

The following example shows an XML document representing the serialization of the integer -559038737.

```
<?xml version="1.0" encoding="UTF-8"?>
<llsd>
  <integer>-559038737</integer>
</llsd>
```

While this example shows the serialization of a binary array of octets containing the values 222, 173, 190 and 239.

```
<?xml version="1.0" encoding="UTF-8"?>
<llsd>
  <binary encoding="base64">3q2+7w==</binary>
</llsd>
```

3.1.2. Serializing Composite Types

Composite types in the XML serialization scheme are represented with 'array' and 'map' elements. Both of these elements may contain elements enclosing simple types or other composite types. Array elements, which represent a collection of values indexed by position, contain a simple list of typed values. Map elements represent a collection of values indexed by a string identifier. They contain a list of key-value pairs where the 'key' element describes the indexing identifier while the value (which follows the 'key' element) is its XML representation.

Note that elements of an array may be of differing types. Also note that composite types may contain other composite types; it is not an error for an array or map to contain another array, map or simple type.

3.1.3. Example of XML LLSD Serialization

This example shows the XML serialization of an array which contains an integer, a UUID and a map.

```
<?xml version="1.0" encoding="UTF-8"?>
<llsd>
 <array>
  <integer>42</integer>
  <uuid>6bad258e-06f0-4a87-a659-493117c9c162</uuid>
  <map>
   <key>hot</key>
   <string>cold</string>
   <key>higgs_boson_rest_mass</key>
   <undef/>
   <key>info_page</key>
   <uri>https://example.org/r/6bad258e-06f0-4a87-a659-493117c9c162</uri>
   <key>status_report_due_by</key>
   <date>2008-10-13T19:00.00Z</date>
  </map>
 </array>
</llsd>
```

3.2. JSON Serialization

LLSD may also be serialized using the JSON [RFC4627] subset of the JavaScript programming language. When serializing LLSD data using JSON, the 'application/llsd+json' media type is used. This specification REQUIRES that LLSD data serialized into a JSON document use UTF-8 character encoding. To allow the serialization of non-composite elements, this specification defines the contents of a JSON-serialized LLSD message in terms of the 'value' non-terminal from RFC 4627 instead of the commonly used 'JSON-text' non-terminal.

The following table lists type conversions between LLSD and JSON:

Undefined LLSD 'Undefined' values are represented by the JSON terminal 'null'.

Boolean LLSD 'Boolean' values are represented by the JSON terminals 'true' and 'false'.

Integer LLSD 'Integer' values are represented by the JSON non-terminal 'number'.

Real LLSD 'Real' values are represented by the JSON non-terminal 'number'.

String LLSD 'String' values are represented by the JSON 'string' non-terminal. Note that this specification inherits JSON's behavior of requiring control characters, reverse solidus and quotation mark characters to be escaped.

UUID LLSD 'UUID' values are represented by a JSON string, and are rendered in the common 8-4-4-4-12 format defined by the 'UUID' non-terminal in RFC 4122 [RFC4122].

Date LLSD 'Date' values are represented by the JSON 'string' non-terminal, the contents of which is a valid ISO 8601 value with years, months, days, hours, seconds and time zone indicator.

URI LLSD 'URI' values are represented by the JSON 'string' non-terminal, the contents of which is a valid URI as defined by RFC 3986 [RFC3986].

Binary LLSD 'Binary' values are represented as a JSON 'array'. That is, they follow the RFC 4627 [RFC4627] 'array' non-terminal whose members are integer numbers representing each octet of the binary array.

Array LLSD 'Array' values are represented by the JSON 'array' non-terminal.

Map LLSD 'Map' values are represented by the JSON 'object' non-terminal. Each key-value pair of the map is represented by the JSON 'member' non-terminal where the LLSD map key is the 'string' prior to the 'name-separator' terminal and the LLSD map value is the 'value' after the 'name-separator' terminal.

LLSD defines additional types over those defined by JSON. The LLSD types UUID, Date and URI are serialized as JSON strings whose contents are generated using the <Type> to String converstion defined in Abstract Type System section above.

3.2.1. Examples of JSON LLSD Serialization

Example 1. The following example shows the JSON encoding of the integer 42. Note that while this serialization does not conform to the 'JSON-text' non-terminal defined in RFC 4627, it does conform to the 'value' non-literal.

```
42
```

Example 2. The following example shows the JSON encoding of the example given in the section above on XML serialization (Section 3.1.2).

```
[
 42,
 "6bad258e-06f0-4a87-a659-493117c9c162",
 {
  "hot": "cold",
  "higgs_boson_rest_mass": null,
  "info_page":
    "https://example.org/r/6bad258e-06f0-4a87-a659-493117c9c162",
  "status_report_due_by": "2008-10-13T19:00.00Z"
 }
]
```

3.3. Binary Serialization

The LLSD Binary Serialization is an encoding syntax appropriate for situations where high message entropy is required or limiting processing power for parsing messages is available.

Encoding LLSD structured data using the binary serialization scheme involves generating tag, (optional) size values, and serialization of simple values. Composite types are serialized by iterating across

all members of the collection, serializing each simple or composite member in turn. For each element in an LLSD structured data object, the following process is used to generate a binary output stream of serialized data:

- A one octet type tag is emitted to the output stream. See the table below for tag octets.
- If the size of the element being serialized is variable (as it will be for strings, URIs, arrays and maps), the size or length of the element is output to the stream as a network-order 32 bit value. Elements of types with fixed lengths such as undefined values, booleans, integers, reals, uuids and dates will not include size information in the output stream.
- Finally, the binary representation of the element is appended to the output stream.

Undefined Undefined values are serialized with a single exclamation point character ('!'). Undefined values append neither size information or data to the output stream.

Boolean True values are serialized with a single '1' character. False values are serialized with a single '0' character. Booleans append neither size information or data to the output stream.

Integer Integer values are serialized by emitting the 'i' character to the output stream followed by the four octets representing the integer's 32 bits in network order.

Real Real values are serialized by emitting the 'r' character to the output stream followed by the eight octets representing the real value's 64 bits in network order.

String String values are serialized by emitting the 's' character to the output stream followed by the string's length in octets represented as a network-order 32 bit integer, followed by the string's UTF-8 encoding.

UUID UUID values are serialized by emitting the 'u' character to the output stream followed by the sixteen octets representing the UUID's 128 bits, with the most significant byte coming first.

Date Date values are serialized by emitting the 'd' character to the output stream followed by the number of seconds since the start of the epoch, represented as a 64-bit real value.

URI URI values are serialized by emitting the 'l' character to the output stream followed by the uri's length in octets represented as a network-order 32 bit integer, followed by the binary representation of the URI.

Binary Binary values are serialized by emitting the 'b' character to the output stream followed by the binary array's length in octets represented as a network-order 32 bit integer, followed by the octets of the binary array.

Array Arrays are serialized by emitting the left square bracket ('[') character, followed by the count of objects in the array represented as a network-order 32 bit integer, followed by each array element in order. Note that compliant implementations MUST preserve the order of array elements.

Map Maps are serialized by emitting the left curly brace ('{') character, followed by the count of objects in the map represented as a network-order 32 bit integer, followed by each key-value element. Map keys are represented as strings except that they use the character 'k' instead of the character 's' as a tag. Note that preserving the order of maps is not REQUIRED.

3.3.1. Example of BINARY LLSD Serialization

The LLSD object given as an example in the section above on XML serialization (Section 3.1.2) would look as follows would it have been serialized using the binary scheme. The following example encodes octets as hexadecimal values.

```
 Offset   Hex Data                  Char Data
-------- ------------------------ -----------
00000000  5B                       '['
00000001  00 00 00 03              '....'
00000005  69                       'i'
00000006  00 00 00 2A              '...*'
0000000A  75                       'u'
0000000B  6B AD 25 8E 06 F0 4A 87  'k.%...J.'
00000013  A6 59 49 31 17 C9 C1 62  '.YI1...b'
0000001B  7B                       '{'
0000001C  00 00 00 04              '....'
00000020  6B                       'k'
00000021  00 00 00 03              '....'
00000025  68 6F 74                 'hot'
00000028  73                       's'
00000029  00 00 00 04              '....'
0000002D  63 6F 6C 64              'cold'
00000031  6B                       'k'
00000032  00 00 00 13              '....'
00000036  68 69 67 67 73 5F 62 6F  'higgs_bo'
0000003E  73 6F 6E 5F 72 65 73 74  'son_rest'
00000046  5f 6d 61 73 73           '_mass'
0000004B  21                       '!'
0000004C  68                       'k'
0000004D  00 00 00 09              '....'
00000051  69 6E 66 6F 5F 70 61 67  'info_pag'
00000059  65                       'e'
0000005A  6C                       'l'
0000005B  00 00 00 3A              '...:'
0000005F  68 74 74 70 73 3A 2f 2F  'https://'
00000067  65 78 61 6D 70 6C 65 2E  'example.'
0000006F  6F 72 67 2F 72 2F 36 62  'org/r/6b'
00000077  61 64 32 35 38 65 2D 30  'ad258e-0'
0000007F  36 66 30 2D 34 61 38 37  '6f0-4a87'
00000087  2D 61 36 35 39 2D 34 39  '-a659-49'
0000008F  33 31 31 37 63 39 63 31  '3117c9c1'
00000097  36 32                    '62'
00000099  68                       'k'
0000009A  00 00 00 14              '....'
0000009E  73 74 61 74 75 73 5F 72  'status_r'
000000A7  65 70 6F 72 74 5F 64 75  'eport_du'
000000AF  65 5F 62 79              'e_by'
000000B3  00 00 00 08              '....'
000000B7  64                       'd'
000000B8  41 D2 3C E6 AC 00 00 00  'A.<.....'
```

4. Interface Description Language

The Linden Lab Interface Description Language (LLIDL) is the used to describe a RESTful interface to remote resources. LLIDL unambiguously defines interfaces independent of serialization schemes. Rather than defining independent XML, JSON and Binary interfaces, resource interfaces are described in terms of in terms of LLIDL. Network entities generating or parsing LLSD messages may use the LLIDL interface descriptions to mechanically generate serialization specific software to manipulate LLSD data. Despite the emphasis on automated parsing, LLIDL has been designed as BOTH human and machine readable.

4.1. Abstract Data Types and Names

LLIDL describes data structures in terms of name-type pairs. LLIDL data structure members are defined by providing the member name, a colon and an simple LLSD type:

```
name : simple_type
```

4.2. Abstract Data Structures

Data structures may be composed using arrays and maps. Arrays are collections of members, accessed with numeric indices. Simple array descriptions are described in LLIDL using the open brace character ('['), one or more simple LLSD types, and a close brace character (']'). Maps are collections of members, accessed by string keys. Map descriptions begin with an open curly brace character ('{'), on or more name-type pairs and a close curly brace character ('}'). Name-Type pairs in map descriptions are separated by commas (','). The first example below defines an array with three real values. The second defines a map with two string members whose names are 'first_name' and 'last_name'.

```
[ real, real, real ]
```

```
{ first_name: string, last_name: string }
```

The form above defines only fixed-length arrays. To define an array of arbitrary length, the ellipsis ("...") is used. The following example defines a list of one or more string values.

```
[ string ... ]
```

Using an ellipsis in an array definition with more than one member describes an arbitrary length array whose members' types are defined by the listed type, each in turn. In the first example below, the first element of the array is an integer while the second is a string. If there are more than two elements in the array, even elements (assuming that array indexes begin with zero (0)) will be integers while odd elements will be strings. Also note that such an array will always contain an even number of members. In the second example, the first element would be a string, the second would be a UUID and the third would be a floating point value. If the array contained more than three elements, starting from the beginning, every third element would be a string; the next would be a UUID while the one following would be a real. An array defined in the second example would always be an multiple of three.

```
[ integer, string ... ]

[ string, uuid, real ... ]
```

4.3. Variant Data Structures

It is often advantageous to represent several different variants of a message. LLIDL defines variants with repeated assignments to the same variant name. In the example below, two variants are defined, the first with two strings, and the second with a number and a string.

```
&exception = {
  class       : string ,
  description : string
}

&exception = {
  class       : int ,
  description : string
}
```

4.4. Variant Discriminators

In the example above, two variants of a structure differ by the type defined for the "class" structure member. Because it is possible for name-value pairs to be absent from the serialization of an LLIDL object, it is often useful to use boolean or string literals to distinguish variants of an object. In the example below the success member is used to identify which variant is being used. When serialized, if the value associated with the success member was true, a compliant parser would know it was not an encoding error for the err_num member to not be present. In other words, it signals that the second variant is in use.

```
&response = {
  success     : false ,
  description : string ,
  err_num     : integer
}

&response = {
  success     : true,
  description : string
}
```

4.5. Resource Description

Defining interfaces is the underlying purpose of LLIDL. Each interface has a name, an input definition and an output definition. They are specified using the following format:

```
%% resource name -> request <- response
```

5. IANA Considerations

In accordance with [RFC5226], this document registers the following mime types:

application/llsd+xml

application/llsd+json

application/llsd+binary

See the MIME Type Registrations section (Section 6) below for detailed information on MIME Type registrations.

6. MIME Type Registrations

This section provides media-type registration applications (as per RFC 4288 [RFC4288].)

6.1. MIME Type Registration for application/llsd+xml

To: ietf-types@iana.org

Subject: Registration of media type application/llsd+xml

Type name: application

Subtype name: llsd+xml

Required Parameters: none

Optional Parameters: none

Encoding Considerations: The Extensible Markup Language (XML) specification allows for the use of multiple character sets. The character set used to encode the body of the message is defined as part of the XML header. If no character set is indicated in the XML header, compliant systems MUST assume UTF-8.

Security Considerations: LLSD XML serialized data contains "plain" text and generally poses no immediate risk to system security of either the sender or the receiver. Still, it is possible for a malicious adversary to include arbitrary binary data in an attempt to exploit specific vulnerabilities (if they exist.) It is the obligation of the receiver of LLSD XML serialized messages to ensure such vulnerabilities are mitigated in a timely fashion.

If sensitive information is to be encoded into a LLSD XML serialized message, it is the responsibility of the transport, network or link layers to ensure the confidentiality, message integrity and origin integrity of the message.

Interoperability Considerations: While it is possible for compliant implementations to specify the use of character sets other than UTF-8, such systems MUST accept UTF-8 input and SHOULD generate UTF-8 output.

Published specification: Linden Lab Structured Data (LLSD) is defined in the internet draft draft-hamrick-llsd-01 [I-D.hamrick-llsd].

Applications that use this media type: Virtual world, tele-presence and content management systems related to "virtual reality" systems.

Additional Information:

Magic Number(s): none

File Extension: lsdx

Macintosh File Type Code(s): TEXT

Person & email address to contact for further information: Meadhbh Hamrick <infinity@lindenlab.com>

Intended Usage: COMMON

Author: IESG

Change Controller: IESG

6.2. MIME Type Registration for application/llsd+json

To: ietf-types@iana.org

Subject: Registration of media type application/llsd+json

Type name: application

Subtype name: llsd+json

Required Parameters: none

Optional Parameters: none

Encoding Considerations: Use of UTF-8 is Mandatory RFC 4627 : The application/json Media Type for JavaScript Object Notation (JSON) [RFC4627] allows the use of UTF-8, UTF-16 and UTF-32. This specification REQUIRES the use of UTF-8.

Security Considerations: Like the application/json media type defined in RFC 4627 [RFC4627], the contents of messages identified with this media type are expected to be passed into ECMAScript's 'eval()' function. RFC 4627 provides a regular expression to ensure that only "safe" characters (i.e. - characters used to describe JSON tokens) are included outside string literal definitions. Users of the application/llsd+json media type are strongly encouraged to use this (or similar) tests

to ensure message safety.

If sensitive information is to be encoded into a LLSD JSON serialized message, it is the responsibility of the transport, network or link layers to ensure the confidentiality, message integrity and origin integrity of the message.

Interoperability Considerations: Note that unlike RFC 4627, this specification REQUIRES the use of UTF-8.

Published specification: This specification.

Applications that use this media type: Virtual world, tele-presence and content management systems related to "virtual reality" systems.

Additional Information:

Magic Number(s): none

File Extension: lsdj

Macintosh File Type Code(s): TEXT

Person & email address to contact for further information: Meadhbh Hamrick <infinity@lindenlab.com>

Intended Usage: COMMON

Author: IESG

Change Controller: IESG

6.3. MIME Type Registration for application/llsd+binary

To: ietf-types@iana.org

Subject: Registration of media type application/llsd+binary

Type name: application

Subtype name: llsd+binary

Required Parameters: none

Optional Parameters: none

Encoding Considerations: LLSD Binary Serialization REQUIRES the use of binary content-transfer-encoding Section 5 of RFC 2045 [RFC2045] describes the binary Content-Transfer-Encoding header field. This specification REQUIRES the use of this header to alert intermediary systems that information being included in the message should be interpreted as binary data with no end-of-line semantics which could be considerably longer than allowed in an RFC 821 transport.

Security Considerations: This serialization format defines the use of tagged binary fields with embedded length information. In the past, similar binary encoding systems have fallen prey to exploits when parsing implementations fail to check for non-sensical lengths. Implementers are therefore strongly encouraged to consider all failure modes of such a system.

If sensitive information is to be encoded into a LLSD JSON serialized message, it is the responsibility of the transport, network or link layers to ensure the confidentiality, message integrity and origin integrity of the message.

Interoperability Considerations: none

Published specification: Linden Lab Structured Data (LLSD) is defined in the internet draft draft-hamrick-llsd-01 [I-D.hamrick-llsd].

Applications that use this media type: Virtual world, tele-presence and content management systems related to "virtual reality" systems.

Additional Information:

Magic Number(s): none

File Extension: lsdb

Macintosh File Type Code(s): LSDB

Person & email address to contact for further information: Meadhbh Hamrick <infinity@lindenlab.com>

Intended Usage: COMMON

Author: IESG

Change Controller: IESG

7. Security Considerations

Security considerations for this specification are, fortunately, either simple or beyond the scope of this document. RFC 3552 [RFC3552] describes several aspects to use when evaluating the security of a specification or implementation. We believe most common security concerns users of this specification will encounter are more appropriately considered as transport, network or link layer issues. Or, as higher level "application security" issues.

This document specifies the content, media type identifiers and content encoding requirements for LLSD. It does not specify mechanisms to transmit LLSD messages between network peers. We believe that many communication security considerations such as confidentiality, data integrity and peer entity authentication are more appropriately the domain of message, transport, network or link layer protocols. Users of this protocol should seriously consider the use Secure MIME, Transport Layer Security (TLS), IPSec or related technologies.

8. References

8.1. Normative References

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3339] Klyne, G., Ed. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, July 2002.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, July 2005.

[RFC4288] Freed, N. and J. Klensin, "Media Type Specifications and Registration Procedures", BCP 13, RFC 4288, December 2005.

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[TR15] Davis, M. and M. Durst, "Unicode Standard Annex #15 : UNICODE NORMALIZATION FORMS", 2008, <http://unicode.org/reports/tr15/>.

[XML2006] Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fourth Edition)", 2006.

8.2. Informative References

[I-D.hamrick-llsd] Brashears, A., Hamrick, M., and M. Lentczner, "Linden Lab Structured Data", 2008.

[ISO8601] "ISO 8601 - Date and Time Formats".

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, July 2003.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

Appendix A. ABNF of Real Values

The following is the Augmented Backus-Naur Form (ABNF) of valid Real values for the purposes of converting strings into real values. ABNF is described in RFC 5234 [RFC5234].

```
real              =  zero
real              =/ negative-infinity
real              =/ negative-zero
real              =/ positive-zero
real              =/ positive-infinity
real              =/ signaling-nan
real              =/ quiet-nan
real              =/ realnumber

negative-infinity =  %x2D.49.6E.66.69,6E.69.74.79   ; "-Infinity"
negative-zero     =  %x2D.5A.65.72.6F               ; "-Zero"
zero              =  %x30.2E.30                     ; "0.0"
positive-zero     =  %x2B.5A.65.72.6F               ; "+Zero"
positive-infinity =  %x2B.49.6E.66.69,6E.69.74.79   ; "+Infinity"
signaling-nan       =  %4E.61.4E.53                   ; "NaNS"
quiet-nan         =  %4E.61.4E.51                   ; "NaNQ"

realnumber        =  mantissa exponent

mantissa          =  ( positive-number [ "." *decimal-digit ])
mantissa          =/ ( "0." *("0") positive-number )

exponent          =  "E" ( "0" / ( [ "-" ] positive-number ) )

positive-number   =  non-zero-digit *decimal-digit

decimal-digit       =  %x30-39
non-zero-digit    =  %x31-39
```

Appendix B. XML Serialization DTD

The following Document Type Definition (DTD) describes the format of LLSD XML Serialization. DTDs are described in the Extensible Markup Language (XML) 1.0 (Fourth Edition) [XML2006] specification.

```
<!DOCTYPE llsd [
<!ELEMENT llsd (DATA)>
<!ELEMENT DATA (SIMPLE|map|array)>
<!ELEMENT
  SIMPLE (undef|boolean|integer|real|uuid|string|date|uri|binary)>
<!ELEMENT KEYDATA (key,DATA)>
<!ELEMENT key (#PCDATA)>
<!ELEMENT map (KEYDATA*)>
<!ELEMENT array (DATA*)>
<!ELEMENT undef (EMPTY)>
<!ELEMENT boolean (#PCDATA)>
<!ELEMENT integer (#PCDATA)>
<!ELEMENT real (#PCDATA)>
<!ELEMENT uuid (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT uri (#PCDATA)>
<!ELEMENT binary (#PCDATA)>

<!ATTLIST string xml:space (default|preserve) 'preserve'>
<!ATTLIST binary encoding CDATA "base64">
]>
```

Appendix C. ABNF of LLIDL

The following is the Augmented Backus-Naur Form (ABNF) of the Linden Lab Interface Description Language (LLIDL). ABNF is described in RFC 5234 [RFC5234].

```
value           =  type / array / map / selector / variant

type            =  "undef"
type            =/ "string"
type            =/ "bool"
type            =/ "int"
type            =/ "real"
type            =/ "date"
type            =/ "uri"
type            =/ "uuid"
type            =/ "binary"
```

```
   array          =  "[" s value-list s "]"
   array          =/ "[" s value-list s "..." s "]"

   map            =  "{" s member-list s "}"
   map            =/ "{" s "$" s ":" s value s "}"

   value-list     = value [ s "," [ s value-list ] ]

   member-list    = member [ s "," [ s member-list ] ]
   member         = name s ":" s value

   selector       =  quote name quote
   selector       =/ "true" / "false"
   selector       =/ 1*digit

   variant        = "&" name


   definitions    = *( s / variant-def / resource-def )

   variant-def    = "&" name s "=" s value

   resource-def   = res-name s res-request s res-response
   res-name       = "%%" s name
   res-request    = "->" s value
   res-response   = "<-" s value


   s              = *( tab / newline / sp / comment )
   comment        = ";" *char newline
   newline        = lf / cr / (cr lf)

   tab            = %x0009
   lf             = %x000A
   cr             = %x000D
   sp             = %x0020
   quote          = %x0022
   digit          = %x0030-0039
   char           = %x09 / %x20-D7FF / %xE000-FFFD / %x10000-10FFFF

   name           = name_start *name_continue
   name_start     = id_start    / "_"
   name_continue  = id_continue / "_" / "/"
   id_start       = %x0041-005A / %x0061-007A ; ALPHA
   id_continue    = id_start / %x0030-0039    ; DIGIT
```

Authors' Addresses

Aaron Brashears
Linden Research, Inc.
945 Battery St.
San Francisco, CA 94111
US

Phone: +1 415 243 9000
Email: aaronb@lindenlab.com

Meadhbh Siobhan Hamrick (editor)
Linden Research, Inc.
945 Battery St.
San Francisco, CA 94111
US

Phone: +1 650 283 0344
Email: infinity@lindenlab.com

Mark Lentczner
Linden Research, Inc.
945 Battery St.
San Francisco, CA 94111
US

Phone: +1 415 243 9000
Email: zero@lindenlab.com