Virtual World Region Agent Protocol
Internet-Draft
Intended status: Standards Track
Expires: January 6, 2011

A. Brashears
M. Hamrick
M. Lentczner

July 5, 2010

VWRAP : Abstract Type System for the Transmission of Dynamic Structured Data

draft-ietf-vwrap-type-system-00

Abstract

This document describes the LLIDL interface description language, the related LLSD abstract type system and three serialization formats for LLIDL messages. LLIDL (pronounced "little") is a language-neutral facility for describing transport independent message flows for RESTful resource access. LLIDL itself is an abstract meta-grammar for producing and recognizing valid request / response messages affecting state change in application layer objects by way of RESTful resource access. It may be used by protocol developers and system deployers to describe the composition of application layer protocol exchanges without adopting transport specific message semantics or programming language specific type semantics. The type behavior of individual message elements is described by the LLSD abstract type system. Abstract LLIDL messages are concretized using one of three defined LLSD serialization schemes. Serialization / deserialization rules are provided in this document for XML, JSON and Binary schemes. This abstract messaging and type system is intended to be used by other specifications to describe application layer protocol exchanges, independent of implementation language or message transport protocol.

Status of this Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

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."

This Internet-Draft will expire on January 6, 2011.

Copyright Notice

Table of Contents

1. Introduction

It is characteristic of modern network services that they are deployed across multiple network hosts. For performance, fault tolerance, ease of deployment or organizational reasons, software and systems implementing network services must now work well in a distributed environment. It is generally believed that such distributed services may be made more robust by making their components "loosely coupled."[Kaye2003] This document describes an interface description language and a related abstract type system used to define interfaces to loosely coupled network services in a programming language, network transport and message serialization independent manner.

The LLIDL interface description language may be used to define protocol exchanges for accessing resources exhibiting characteristics of the Representational State Transfer (REST) architecture style. [Fielding2000] LLIDL describes abstract interfaces intended to be reified over HTTP [RFC2616] or HTTPS [RFC2817]. LLIDL resource definitions describe the structure of data provided in an access request, the structure of the data in the access' response and the HTTP verbs which may be used to access the resource.

The LLSD abstract type system defines nine simple types (Undefined, Boolean, Integer, Real, String, UUID, Date, URI and Binary) and two composite types (Array and Map.) This system provides a programming language independent framework for describing type semantics of elements in LLIDL messages. Three serialization schemes are defined by this document: XML, JSON and Binary. These schemes are used to concretize LLSD data into octet streams for transmission over a data network. Each serialization scheme has a related MIME content type definition, allowing compliant applications to identify the specific serialization scheme used.

LLIDL and LLSD form an abstract system for reasoning about application layer exchanges without having to repeatedly reference the details of the transport used to deliver messages. Other specifications use LLIDL and LLSD to describe the content of RESTful resource access. This document describes how resource accesses are reified as HTTP(S) protocol exchanges. LLIDL is intended to separate the semantics of application messages from the details of the protocol that carries them. It gives system deployers a tool for succinctly defining application layer exchanges.

The LLSD serialization schemes describe how simple and composite types are converted into an octet stream and provides guidelines for transmission across a network. It does not describe the concretization of abstract LLSD messages into programming language

constructs.

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. The LLSD Abstract Type System

The LLSD abstract type system describes the semantics of data passed between two network hosts. 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 anticipated 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. Nonetheless, systems that transmit 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 created.

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 arbitrary 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. The LLIDL Interface Description Language

3.1. Interfaces and Resources

A LLIDL "Interface" is comprised conceptually of collection of zero or more related resources and named type definitions. The LLIDL grammar defines an "Interface Definition" as being zero or more comments, named type definitions or resource definitions.

A LLIDL "Resource" represents information or state maintained by a remote system, accessed via HTTP(S). A "Resource Definition" is the grammatical construction used to represent a resource. Resources are partitioned into "method access classes" based on the HTTP verbs used to access them. Method access classes include: "GET", "GET/PUT", "GET/PUT/DELETE" and "POST". Method access classes are notated in the LLIDL grammar using "Method Access Delimeters": "<<" for GET, "<>" for GET/PUT, "<x>" for GET/PUT/DELETE and POST is notated with the pair of strings "->" and "<-".

Resource definitions also include a message body defining the structure of requests and responses. GET, GET/PUT and GET/PUT/DELETE

resources define a single message body following the method access delimiter. POST resources define two message bodies. The first follows the "->" delimiter and represents the request. The second follows the "<-" delimiter and represents the response.

A single simple type definition or "flat" map may be defined in conjunction with the resource that describes the contents of arguments to be placed in the query string of the request. A flat map is a map containing only simple types (i.e. - it does not contain arrays or maps.)

A resource definition has the format:

```
'%%' <resource-name> [ '??' <query-body> ]
   <resource-delimeter> <message-body> [ <- <message-body> ]
```

The resource-name identifies the resource (not the URL at which it is located.) Resource-names are strings that may contain alphabetic characters, numbers, the slash character ('/') and the underbar character ('_').

3.2. Simple Types

LLIDL uses the nine simple types from LLSD to define the type behavior of scalar elements in a resource. These types are undefined, boolean, integer, real, string, UUID, URI, date and binary. They are declared in LLIDL with different identifiers that are (respectively): undef, bool, int, real, string, uuid, uri, date and binary. Note that the undefined, boolean and integer types are declared using a more compact textual description of the type.

3.3. Composite Types

Composite Types are resource elements that contain more than one value. LLIDL uses the two composite types from LLSD: array and map.

3.3.1. Arrays

Arrays represent a sequence of simple types. Each element in an array is accessed by an ordinal value. An array declaration begins with the open bracket character ('[') and ends with the close bracket character (']'). Within the definition of an array, comma delimited type declarations describing the type of each element are given.

The format for an array declaration is:

```
'[' <type> [ ',' <type> ] ... ']'
```

The following example declares a five element array whose elements' types are three integers, a string and a URI:

```
[ int , int , int , string , uri ]
```

LLIDL arrays may also be of indeterminate length. The ellipsis trigraph ("...") appended to the end of a sequence of types in an array declaration indicates the previously defined sequence of types is repeated indefinitely.

The following examples describe (first) an arbitrary lengthed array comprising of strings and (second) an arbitrary lengthed array comprising of three real values followed by a string:

```
[ string , ... ]

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

Note that the ellipsis trigraph indicates that the entire sequence is repeated, not only the last element.

It is acceptable for an array to contain composite types like arrays or maps. The following example describes an array whose elements are an array of three real values and a string:

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

3.3.2. Maps

Maps are collections of simple types whose elements are accessed via alphanumeric strings. Maps declarations begin with the open brace character ('{') and end with the close brace character ('}'). Within the map declaration are a sequence of comma delimited map entries. Map entries are comprised of a map entry name and a map entry type, separated by a colon character (':'). Map entry names are alphanumeric strings intended to be indicative of their function in the resource definition.

The format of a map definition is:

```
'{' <map-entry-name> ':' <map-entry-type>
    [ ',' <map-entry-name> ':' <map-entry-type> ]  ... '}'
```

The following example describes a map with three elements named: name, position and current_balance. The name entry is declared as a string, the position entry is declared as an array with a string and three real values, and the current_balance entry is declared as an integer.

```
{
  name : string,
  position : [ string , real , real , real ],
  current_balance : int
}
```

As implied by the previous example, it is perfectly acceptable for a map to contain entries whose types are maps and arrays.

It is also possible to define a map in which map entry names that are explicitly unknown at the time a resource is defined. For example, a service may wish to produce or consume a map whose keys come from user data such as stock ticker symbols, avatar names or the names of regions in a virtual world. It is impractical to attempt to define the complete set of possibilities in these cases, so LLIDL allows the resource developer to specify that map names may come from data known only at the time the resource is accessed.

The dollar character ('$') is used to specify a map whose entries' names are determined after the resource is defined and deployed. A map with "deferred entry names" is one in which this situation occurs. Such maps are defined with a single entry whose name is the dollar character and a single type.

The following example shows a map with "deferred entry names" whose map entry types are all URIs.

```
{ $ : uri }
```

At most one deferred entry name specifier (i.e. - one dollar sign) is allowed in a map. A map defined with a deferred entry name specifier may contain no other defined entries.

Deferred entry names do not signify that a later specification will complete the definition of the resource, but that the map's entries' names cannot be determined before the resource is accessed.

3.4. Named Types

LLIDL defines a named type feature. This feature allows a resource developer to define a single alphanumeric symbol that represents a complete type definition. The ampersand ('&') character is used in both the definition and reference of a named type. To define a named type, the following format is used:

'&' <named-type-symbol> '=' <named-type-value>

The named type symbol must be a valid alphanumeric symbol consisting

of upper and lower case letters, numbers, the slash character ('/') or the underbar ('_'). The named type value must be a valid type definition.

The following examples are all valid named type definitions:

```
&example  = string

&info     = { name : string, id : uuid }

&position = [ real, real, real ]
```

Named types are referenced using only the ampersand and a symbol. The following example describes two resources whose response bodies are defined using a named type:

```
&error = { errno : int, desc : string, more : uri }

%% session/search -> string <- &error

%% session/continue  -> uuid   <- &error
```

3.5. Variant Type Definitions

It may be advantageous for a resource to accept more than one form. In this case, a variant type definition may be used. Variant type definitions are defined using the named type feature to define a named type using the same named type symbol for multiple named type definitions.

For example, the following resource defines a response with two forms. The first describes a success condition while the second an error.

```
&request = {
     name   : string,
     secret : binary
}

&response = {
  success    : true,
  session_id : uuid
}

&response = {
  success    : false,
  error      : int,
  next       : uri
}

%% session/establish -> &request <- &response
```

In this example, the first named type (whose named type symbol is 'request') is a simple named type. It is later used in a resource definition to represent the contents of a request to the resource. The second and third named types define a variant. That is, the named type symbol is used more than once. The 'response' variant defined in this example indicates that the response from the resource access will be one of the two 'response' forms.

A "selector" may be used to help determine which variant should be used. A selector is a literal value included in a map entry that appears in each variant. In the example above, the map entry named 'success' has two literal values in the two variants in which it is defined. It is possible to have multiple selectors in a map variant, and the same literal value may be reused.

4. 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.

4.1. 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).

When encoding binary data using RFC 4648 [RFC4648], characters outside the base alphabet are explicitly allowable and should be ignored.

4.1.1. 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>
```

4.1.2. 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.

4.1.3. 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>
```

4.2. JSON Serialization

LLSD may also be serialized using the JSON [ECMA262r5] subset of the JavaScript programming language. When serializing LLSD data using JSON, the 'application/llsd+json' media type is used. The grammar of LLSD objects serialized using the JSON serialization MUST conform to the JSONText production.

The following table lists type conversions between LLSD and JSON:

Undefined LLSD 'Undefined' values are represented by the JSON non-terminal 'JSONNullLiteral'.

Boolean LLSD 'Boolean' values are represented by the JSON non-terminal 'JSONBooleanLiteral'.

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

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

String LLSD 'String' values are represented by the JSON 'JSONString' 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 'JSONArray'. That is, they follow the ECMA-262 [ECMA262r5] 'JSONArray' non-terminal whose members are integer numbers representing each octet of the binary array.

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

Map LLSD 'Map' values are represented by the JSON 'JSONObject' non-terminal. Each key-value pair of the map is represented by the JSON 'JSONMember' non-terminal where the LLSD map key is the 'JSONString' prior to the name separator terminal (':') and the LLSD map value is the 'JSONValue' after the name separator.

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 conversion defined in Abstract Type System section above.

4.2.1. Examples of JSON LLSD Serialization

Example 1. The following example shows the JSON encoding of the integer 42.

```
42
```

Example 2. The following example shows the JSON encoding of the example given in the section above on XML serialization (Section 4.1.2).

```
[
 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"
 }
]
```

4.3. 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, and adding a closing tag. 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.

o 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. Following the elements in

the array, a single octet closing tag is appended to the enclosing. The closing tag for arrays is a single right square bracket (']').

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. Following the elements in the map, a single octet closing tag is appended to the enclosing. The closing tag for arrays is a single right curly brace ('}').

4.3.1. Example of BINARY LLSD Serialization

The LLSD object given as an example in the section above on XML serialization (Section 4.1.2) 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.<.....'
```

5. IANA Considerations

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

application/llidl

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/llidl

To: ietf-types@iana.org

Subject: Registration of media type application/llidl

Type name: application

Subtype name: llidl

Required Parameters: none

Optional Parameters: none

Encoding Considerations: LLIDL may be used with any character set that encodes character points identical to ASCII for the first 127 characters. Compliant systems SHOULD use UTF-8 and if no character set is indicated, UTF-8 MUST be assumed.

Security Considerations: LLIDL interface descriptions contain "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 to ensure such vulnerabilities are mitigated in a timely fashion

In the unlikely event that sensitive information is to be expressed as an LLIDL interface, 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: The grammar of LLIDL is defined in the internet draft draft-ietf-vwrap-type-system-00 [I-D.ietf-vwrap-type-system].

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: llidl

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+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: The LLSD XML Serialization is defined in the internet draft draft-ietf-vwrap-type-system-00 [I-D.ietf-vwrap-type-system].

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

6.3. 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: This specification requires that LLSD objects encoded using the JSON serialization scheme encode their data using Unicode. It is assumed that the transport will carry meta-data describing the character encoding used (UTF-8, UTF-16, UTF-32, etc.) The UTF-8 character encoding is assumed if a character encoding is not specified.

Security Considerations: The contents of messages identified with this media type are expected to be passed into ECMAScript's 'parse()' function. RFC 4627 [RFC4627] 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: none

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

6.4. 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 nonsensical 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: The LLSD binary serialization is defined in the internet draft draft-hamrick-llsd-01 [I-D.ietf-vwrap-type-system].

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

[ECMA262r5] ECMA International, "Standard ECMA-262, 5th Edition : ECMAScript Language Specification", December 2009, <http://www.ecma-international.org/publications/standards/Ecma-262.htm>.

[I-D.ietf-vwrap-type-system] Brashears, A., Hamrick, M., and M. Lentczner, "VWRAP : Abstract Type System for the Transmission of Dynamic Structured Data", July 2010.

[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.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC2817] Khare, R. and S. Lawrence, "Upgrading to TLS Within HTTP/1.1", RFC 2817, May 2000.

[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.

[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

[Fielding2000] University of California, Irvine, "Architectural Styles and the Design of Network-based Software Architectures", 2000.

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

[Kaye2003] The Conversations Network, "Loosely Coupled : The Missing Pieces of Web Services", 2003.

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

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

[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.

```
<!ELEMENT llsd
  (undef|boolean|integer|real|string|uuid|date|uri|binary|array|map)*>
<!ELEMENT undef EMPTY>
<!ELEMENT boolean (#PCDATA)>
<!ELEMENT integer (#PCDATA)>
<!ELEMENT real (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT uuid (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT uri (#PCDATA)>
<!ELEMENT binary (#PCDATA)>
<!ELEMENT array
  (undef|boolean|integer|real|string|uuid|date|uri|binary|array|map)*>
<!ELEMENT map
  (key,(undef|boolean|integer|real|string|uuid|date|uri|binary|array
  |map))*>
<!ELEMENT key (#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 LLIDL Interface Description Language. ABNF is described in RFC 5234 [RFC5234].

```
llidl           = *( s  / resource-def / variant-def )

resource-def    = res-name s res-transaction
res-name        = "%%" s name
res-transaction = res-get / res-getput / res-getputdel / res-post
res-get         = "<<" s value
res-getput      = "<>" s value
res-getputdel   = "<x>" s value
res-post        = res-request s res-response
res-request     = "->" s value
res-response    = "<-" s value

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

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

   type           =  %x75.6E.64.65.66     ; "undef"
   type           =/ %x73.74.72.69.6E.67  ; "string"
   type           =/ %x62.6F.6F.6C        ; "bool"
   type           =/ %x69.6E.74           ; "int"
   type           =/ %x72.65.61.6C        ; "real"
   type           =/ %x64.61.74.65        ; "date"
   type           =/ %x75.72.69           ; "uri"
   type           =/ %x75.75.69.64        ; "uuid"
   type           =/ %x62.69.6E.61.72.79  ; "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       =/ %x74.72.75.65        ; "true"
   selector       =/ %x66.61.6C.73.65     ; "false"
   selector       =/ 1*digit

   variant        = "&" name


   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
```

Appendix D. Glossary

Access See Resource Access.

Array An array is a collection in which elements are accessed by numeric index. By default arrays are fixed length, but a trailing ellipsis in an array definition denotes an array of indeterminate length.

Array Definition An array definition is a feature of the LLIDL grammar used to denote arrays of fixed or indeterminate size. An array definition is a comma delimited sequence of type definitions describing the type of each array element.

Composite Type A composite type in LLIDL and LLSD is an abstract representation of an array or a map.

Deferred Entry Name A map defined with a single "Deferred Entry Name Specifier" (i.e. - the dollar sign), signifies that the map entry's names will be determined at the time the resource is accessed, not when the resource is defined.

Defined Type A defined type is a feature of the LLIDL grammar used to represent one of the eleven (11) predefined types. Defined types are in contrast to literals and map variants. The eleven predefined types are: undefined, boolean, integer, real, date, uuid, uri, string, binary, array and map.

GET (Method Access) The GET method access, denoted in LLIDL with the double less-than digraph ("<<") indicates a given resource should be accessed via the HTTP GET verb. In LLIDL, a single message body definition comes after the double less-than digraph and indicates the composition of the message the client should expect from the server.

GET/PUT (Method Access) The GET/PUT method access, denoted in LLIDL with the less-than / greater-than digraph ("<>") indicates a given resource should be accessed via either the HTTP GET or HTTP PUT verbs. In LLIDL, a single message body definition comes after the less-than / greater-than digraph and indicates the composition of the message the client should expect from the server (if the GET HTTP verb is used) or the composition of the message the server should expect from the client (if the PUT HTTP verb is used.)

GET/PUT/DELETE (Method Access) The GET/PUT/DELETE method access, denoted in LLIDL with the less-than / x / greater-than trigraph ("<x>") indicates a given resource should be accessed via either the HTTP GET, HTTP PUT or HTTP DELETE verbs. In LLIDL, a single message body definition comes after the less-than / x / greater-than trigraph and indicates the composition of the message the client should expect from the server (if the GET HTTP verb is used) or the composition of the message the server should expect from the client (if the PUT HTTP verb is used.)

Interface An LLIDL Interface is a collection of zero or more resources and any variant record definitions they reference.

Literal Values in LLIDL resource definitions usually represent types. A literal value may be used when an element in a message body is to be fixed to a particular value. Literals are in contrast to defined types or map variants.

LLIDL LLIDL is an interface description language for describing RESTful resources accessed via HTTP or HTTPS.

LLSD LLSD is an abstract type system used to describe the structure of data in LLIDL Resource Definitions.

Map A map is a collection in which elements are accessed by a string key.

Map Definition A map definition is a feature of the LLIDL grammar used to denote maps. Map definitions are comprised of zero or more comma delimited map entries.

Map Entry A map entry is a component of a map definition and is composed of a key name and a type definition separated by a colon.

Map Variant See Variant Map.

Method Access A method access is a feature of the LLIDL grammar used to describe which set HTTP verbs a client should use to access a resource. Method access classes include 'GET', 'GET/PUT', 'GET/PUT/DELETE' and 'POST'.

Message Body A message body is a feature of the LLIDL grammar that describes the contents of a message flowing between a client and a server. A message body is the type definition that describes completely the structure of a message flowing between systems.

Named Type A named type is a developer declared name for a type, array or map definition.

POST (Method Access) The POST method access, denoted in LLIDL with the hyphen / greater-than ("->") and less-than / hyphen ("<-") digraphs, indicates a given resource should be accessed via the HTTP POST verb. In LLIDL, a message body definition comes after both digraphs. The message body definition following the first digraph indicates the composition of the message the client should POST to the resource's URL while the second message body definition describes the response the client should expect from the server.

Resource A Resource is an abstract representation of information or state maintained by a remote process, potentially on a remote host. LLIDL may be used to describe the structure of a resource and resources may be accessed by sending and receiving messages serialized using one of the three serialization schemes via HTTP(S).

Resource Access The act of accessing a RESTful resource.

Resource Definition An LLIDL Resource Definition is a statement in the LLIDL language describing a single RESTful resource exported by a remote service. Resource definitions include a resource class, optional query parameters, a method access indicating which HTTP verbs are acceptable to the service, the structure of the resource access' request and/or the structure of the resource access' response. Resource definitions may be used along with a serialization scheme to format or parse a resource request or response.

Resource Class A Resource Class is the textual identifier associated with a resource. It is used to uniquely identify a resource in an interface.

Selector See Variant Selector.

Selector Literal A literal used to identify which variant in a map variant should be expected is a "selector literal."

Serialization Scheme A serialization scheme defines rules used to convert a data structure into an octet stream suitable for transmission across a network. Three schemes are defined in this document: XML, JSON and Binary.

Simple Type In LLSD and LLIDL, a "simple type" is a defined type that is not a collection. It is one of: undefined, boolean, integer, real, date, uuid, uri, string or binary.

Type Definition A type definition is a feature of the LLIDL grammar used to declare the type of a data element in a message body. It may be a literal, a defined type or a variant map.

Variant Definition A map used as one of several options in a variant map.

Variant Map A variant type definition in which a map is used for one of the variants. Variant maps may include a selector to assist in matching the most appropriate variant.

Variant Selector A map entry whose value is set as a literal. Used to determine which variant definition of a variant map is to be used.

Variant Type Definition A type definition comprised of more than one definition. Variant type definitions are defined using the named type feature of LLIDL.

Appendix E. Acknowledgements

The authors gratefully acknowledge the contributions of: Lora Baines, Alan Bradley, Suzy Deffeyes, Morgaine Dinova, Kevin Flynn, Valentyn Gatsuk, Walter Gibbs, John Hurliman, Dave Huseby, Charles Krinke, Jennifer Leech, David Levine, Steven Lisberger, Dan Olivares, Catherine Pfeffer, Jon Watte and Ryan Williams.

Authors' Addresses

Aaron Brashears

Meadhbh Siobhan Hamrick
P.O. Box 783
Boulder Creek, CA 95006
US

Phone: +1 650 283 0344
Email: OhMeadhbh@gmail.com

Mark Lentczner