Fledge Message Serialization (second draft)

Tom Gibara

May 2007, last revised June 2008

1 Introduction

This document defines a binary message encoding that has been designed for the Fledge distributed computing network. It has been designed to (in order of priority):

1. provide a compact representation of an encoded message
2. permit efficient implementation
3. be Java friendly without constraining its implementation in other languages
4. support null values efficiently
5. allow a degree of extensibility within established vocabularies.

A number of different technologies are already available that meet some of these goals, unfortunately none meet them all. The software investigated included:

Hessian describes itself as a "simple binary protocol for connecting web services". Of all the possible software investigated, this came closest to suitability. Because the Hessian data format does not have explicit types, each value must be explicitly tagged by its type, this can amount to a significant overhead in the message transmission size.

Java Serialization is a well established and well performing technology but one that is not accessible outside the Java platform.

XStream a well established XML serialization library that can produce readable markup, devoid of Java-only constructs. Unfortunately XML is too verbose for the Fledge network transmissions; this objection naturally bars all XML based serialization mechanisms.

BinaryNotes is an ASN.1 parsing library and compiler. Unfortunately, as with all other ASN.1 libraries for Java, it generates Java source code from an ASN.1 definition. This renders it unsuitable for the Fledge framework since any application on the network would be forced to use the BinaryNotes compiler - an unacceptable constraint.

2 Overview

The message encoding is a straightforward byte-aligned encoding of atomic values, collections and tuples. It separates the encoding into both a type and a representation. The type can be rich - declaring all of the fields of information that are to follow, or very sparse - providing nothing more than a single id from which the decoder is expected to deduce the data structure. The representation contains all of the values that comprise the message data. This arrangement provides for application-level trade-offs to be made concerning compactness against comprehensibility.

The encoding described below currently supports 9 atomic types and 2 collection types. This could be expanded up to 16 atomic types and 8 collection types.

Unless otherwise stated, all encodings are big-endian (that is, MSB are stored first in the bit stream) and all signed values are 2's complement.

3 Atoms

Atoms are the building-blocks of encoding. They have been chosen to provide a superset of the Java primitive types, in addition to which useful universal types have been added.[1]

BOOLEAN	Encoded as 8 bit value: 0 or -1
BYTE	Encoded as 8 bit signed value
SHORT	Encoded as 16 bit signed value
INTEGER	Encoded as 32 bit signed value
LONG	Encoded as 64 bit signed value
FLOAT	Encoded as a IEEE 32 bit float
DOUBLE	Encoded as an IEEE 64 bit float
CHARACTER	Encoded as an 8/16 bit UTF-8 value
TIMESTAMP	Encoded as a 64 bit signed value in machine epoch time

4 Collections

All collections are homogenous (meaning that a single collection can only contain values of identical type). As a consequence, some Java collections cannot be modelled; this is a concious trade-off between simplicity and efficiency on the one hand and applicability and ease-of-use on the other.

[1]At present there is only one, the timestamp. Consideration is being given to providing support for arbitrary precision numbers (`BigInteger` and `BigDecimal`).

The size of a collection of values are encoded using variable byte lengths according to the binary pattern given below. The intention is to ensure that small collections are encoded efficiently without unecessarily constraining the greatest possible size of collections.

0xxxxxxx One byte for lengths expressible in no more than 7 bits.

10xxxxxx xxxxxxxx Two bytes for lengths expressible in no more that 14 bits

110xxxxx xxxxxxxx xxxxxxxx Three bytes for lengths expressible in no more that 21 bits

1110xxxx xxxxxxxx xxxxxxxx xxxxxxxxx Four bytes for lengths expressible in no more that 28 bits

It follows that the maximum size of any collection is $2^{28} - 1$.

- List
  Encoded as length (the number of values in the list) followed by element data
- Bag
  Encoded as size (the number of key/value pairs in the map) followed by interleaved keys and values.

There is no restriction on lists (*resp.* bags) that requires unique values (*resp.* keys). Where lists (*resp.* maps) are used to encode sets (*resp.* maps) it is the responsibility of the encoder/decoder to ensure that such constraints are met.

5 Tuples

Tuples consist of a fixed number of fields, any combination of which may be null. The encoding defined in this document *does not* provide anything like a labelled tuple (a record or object). The organisation of record or object fields into an unambiguous list of fields is expected to be performed as part of the language specific mapping performed during encoding and decoding.

Tuple values are recorded in two parts. A list of values, one for each element of the tuple, is preceeded by a packed bit array[2] in which each bit indicates whether the corresponding field is null. This array is referred to as the *nullary.* Only non-null values follow the nullary. Because null fields are associated with tuples only, collections and atoms cannot by themselves encode null values.

[2]The bit array is is padded with zeros to make its length divisible by 8. The MSB corresponds to first field.

6 Types

Types provide the mechanism by which the structure of the transmitted data is communicated to the receiver. At its simplest, the type of a message consists of a single numeric id which, in the context of the communication, identifies the format of the message being sent. At the other extreme, the sender may choose to send all of the information needed to correctly read the transmitted data;[3] Such transmissions are nevertheless typically very compact.

- Atomic types are allocated a number:

0xE0	Boolean
0xE1	Byte
0xE2	Short
0xE3	Integer
0xE4	Long
0xE5	Float
0xE6	Double
0xE7	Character
0xE8	Timestamp

- Collection types are encoded as:

0xF0	List - followed by the element type
0xF1	Bag - followed by the key type then the value type

- Tuple types are encoded using an id. The id may be *public* which means that is related to an established vocabulary, or *synthetic* which means that the tuple has no preferred representation outside of its encoding within the message.

  Public tuple ids are encoded using the even integers, synthetic ids are encoded using the odd integers. The first 16 odd numbers are reserved to represent tuples consisting of only one atomic value; such tuples are referred to as "atomic tuples" and are useful for encoding nullable primitives:

0x01	Boolean Tuple
0x03	Byte Tuple
0x05	Short Tuple

[3]This message serialization format, by design, *does not* transmit the information needed to correctly interpret the transmitted data in the sense of providing human-readable or object-mapping data. This is expected to be done by extensions to this protocol (eg. in a header), out-of-band transmissions, or otherwise pre-established.

0x07	Integer Tuple
0x09	Long Tuple
0x0B	Float Tuple
0x0D	Double Tuple
0x0F	Character Tuple
0x11	Timestamp Tuple

The id is encoded using 1,2 or 4 bytes as:

0x00-0x7F	for ids not exceeding 7 bits
0x80-0xBF	followed by the 8 least significant id bits, for ids not exceeding 14 bits
0xC0-0xDF	followed by the 24 least significant id bits, for ids not exceeding 29 bits

If this is the first instance of the type within the current message and the type is both unreserved and synthetic, or public but defined, (ie. the type has not been yet been defined but should be), this is followed by:

- number of fields in the tuple
- the type of each field

This additional information is optional if the tuple id is a public one. If omitted, it is assumed that the decoder will know the type's structure based on the public id within the context of the message vocabulary.

7 Message Encoding

Every message is assumed to be in the context of a vocabulary that has been established external to the message. In this context, each message encoding consists of two parts:

1. a type
   The type is written in "structural order". This is the natural ordering of fields that arises from structural recursion on the type.
2. a realization
   The realization of a type consists of values in the structural order of that type.

8 Java Mapping

Atomic values are naturally mapped to their corresponding Java types (primitives in most cases). The atomic tuples also have a standard representation the type system. This is summarized in the table below.

Atomic Type	Java Type	Java Type (tuplized)
BOOLEAN	`boolean`	`java.lang.Boolean`
BYTE	`byte`	`java.lang.Byte`
SHORT	`short`	`java.lang.Short`
INTEGER	`int`	`java.lang.Integer`
LONG	`long`	`java.lang.Long`
FLOAT	`float`	`java.lang.Float`
DOUBLE	`double`	`java.lang.Double`
CHARACTER	`char`	`java.lang.Character`
TIMESTAMP	`long`	`java.util.Date`

All Java collection types (with the exception of maps) and Java array types are encoded using LIST. Java arrays provide the default representation for decoding LIST values.

Instances of `java.util.Map` are encoded using BAG and also provide the default representation.

The default representation of a tuple is a Java object array.

A `java.lang.String` is encoded as a character array.

9 Future Considerations

The following amendments and extensions are under consideration:

- The introduction of a compact boolean array encoding. Whether this is worth the additional complexity of its implementation given that applications can use byte arrays is questionable.
- Provision of a compact encoding for small numbers.
- Support for types equivalent to Java's `BigInteger` and `BigDecimal` classes.
- Reference values; that is, the ability to share repeated values in a realization. This is not intended to provide object-graph semantics, but to allow for efficient data transmission.
- Allowing collection types to defer stating their size, so that collections of an indeterminate size can be streamed; possibly by 'chunking'.
- Non-homogenous collection types.

10 Document History

- First published May 2007
- Corrections and clarifications made June 2008