Thrift specification - Remote Procedure Call

Erik van Oosten <e.vanoosten@grons.nl>

Revision History
Revision 1.0 2016-09-27 EVO
Initial version v1.1, 2016-10-05: Corrected integer type names. Small changes to section headers.

Table of Contents

The missing specification.

1. Introduction

Thrift is a RPC mechanism that easily blends in with your code. It has a wonderful transport protocol that stays backward and forward compatible without the security pitfalls brought by seriapalooza [https://www.contrastsecurity.com/security-influencers/serialization-must-die-act-1-kryo].

This document specifies the so far undocumented thrift RPC message exchange and the wire encoding of those messages in the binary protocol and the more modern compact protocol. Then the binary protocol and compact protocol are compared. Finally it describes the framed vs. unframed transport.

For background on Thrift see the Thrift white paper (pdf) [https://thrift.apache.org/static/files/thrift-20070401.pdf].

This document is for Thrift implementers. Thrift users should read the thrift documentation [https://thrift.apache.org/] and the missing thrift guide [https://diwakergupta.github.io/thrift-missing-guide/].

The information here is based on research in the Java implementation in the Apache thrift library (version 0.9.1 and 0.9.3) and THRIFT-110 A more compact format [https://issues.apache.org/jira/browse/THRIFT-110]. Other implementation however, should behave the same.

Feedback and contributions to this specifications are very welcome. You can find the source code [https://github.com/erikvanoosten/thrift-missing-specification] on GitHub.

There is also a PDF [http://erikvanoosten.github.io/thrift-missing-specification/thrift-rpc-missing-specification.pdf] version of this document.

2. Thrift Remote Procedure Call Message exchange

Both the binary protocol and the compact protocol assume a transport layer that exposes a bi-directional byte stream, for example a TCP socket. Both use the following exchange:

1. $\Rightarrow$ Client sends a `Message` (type `Call` or `Oneway`). The TMessage contains some metadata and the name of the method to invoke.
2. $\Rightarrow$ Client sends method arguments (a struct defined by the generate code).
3. $\Leftarrow$ Server sends a `Message` (type `Reply` or `Exception`) to start the response.
4. $\Leftarrow$ Server sends a struct containing the method result or exception.

The pattern is a simple half duplex protocol where the parties alternate in sending a `Message` followed by a struct. What these are is described below.

Although the standard Apache Thrift Java clients do not support pipelining (sending multiple requests without waiting for an response), the standard Apache Thrift Java servers do support it.

2.1. Message

A **message** contains:

- *Name*, a string.
- *Message type*, a message types, one of `Call`, `Reply`, `Exception` and `Oneway`.
- *Sequence id*, a signed i32 integer.

The **sequence id** is a simple message id assigned by the client. The server will use the same sequence id in the message of the response. The client uses this number to detect out of order responses. Each client has an i32 field which is increased for each message. The sequence id simply wraps around when it overflows.

The **name** indicates the service method name to invoke. The server copies the name in the response message.

When the **multiplexed protocol** is used, the name contains the service name, a colon (`:`) and the method name. The multiplexed protocol is not compatible with other protocols.

The **message type** indicates what kind of message is sent. Clients send requests with messages of type `Call` or `Oneway` (step 1 in the protocol exchange). Servers send responses with messages of type `Exception` or `Reply` (step 3).

Type `Reply` is used when the service method completes normally. That is, it returns a value or it throws one of the exceptions defined in the Thrift IDL file.

Type `Exception` is used for other exceptions. That is: when the service method throws an exception that is not declared in the Thrift IDL file, or some other part of the Thrift stack throws an exception. For example when the server could not encode or decode a message or struct.

In the Java implementation (0.9.3) there is different behavior for the synchronous and asynchronous server. In the async server all exceptions are send as a `TApplicationException` (see *Response struct* below). In the synchronous Java implementation only (undeclared) exceptions that extend `TException` are send as a `TApplicationException`. Unchecked exceptions lead to an immediate close of the connection.

Type `Oneway` is only used starting from Apache Thrift 0.9.3. Earlier versions do *not* send messages of type `Oneway`, even for service methods defined with the `oneway` modifier.

When client sends a request with type `Oneway`, the server must *not* send a response (steps 3 and 4 are skipped). Note that the Thrift IDL enforces a return type of `void` and does not allow exceptions for oneway services.

2.2. Request struct

The struct that follows the message of type `Call` or `Oneway` contains the arguments of the service method. The argument ids correspond to the field ids. The name of the struct is the name of the method with `_args` appended. For methods without arguments an struct is sent without fields.

2.3. Response struct

The struct that follows the message of type `Reply` are structs in which exactly 1 of the following fields is encoded:

- A field with name `success` and id `0`, used in case the method completed normally.
- An exception field, name and id are as defined in the `throws` clause in the Thrift IDL's service method definition.

When the message is of type `Exception` the struct is encoded as if it was declared by the following IDL:

```
exception TApplicationException {
  1: string message,
  2: i32 type
}
```

The following exception ``type``s are defined in the java implementation (0.9.3):

`0`, unknown

used in case the type from the peer is unknown.

`1`, unknown method

used in case the method requested by the client is unknown by the server.

`2`, invalid message type

no usage was found.

`3`, wrong method name

no usage was found.

`4`, bad sequence id

used internally by the client to indicate a wrong sequence id in the response.

`5`, missing result

used internally by the client to indicate a response without any field (result nor exception).

`6`, internal error

used when the server throws an exception that is not declared in the Thrift IDL file.

`7`, protocol error

used when something goes wrong during decoding. For example when a list is too long or a required field is missing.

`8`, invalid transform

no usage was found.

`9`, invalid protocol

no usage was found.

`10`, unsupported client type

no usage was found.

Struct

A **struct** is a sequence of zero or more fields, followed by a stop field. Each field starts with a field header and is followed by the encoded field value. The encoding can be summarized by the following BNF:

```
struct        ::= ( field-header field-value )* stop-field
field-header  ::= field-type field-id
```

Because each field header contains the field-id (as defined by the Thrift IDL file), the fields can be encoded in any order. Thrift's type system is not extensible; you can only encode the primitive types and structs. Therefore it is also possible to handle unknown fields while decoding; these are simply ignored. While decoding, the field type can be used to determine how to decode the field value.

Note that the field name is not encoded so field renames in the IDL do not affect forward and backward compatibility.

The default Java implementation (Apache Thrift 0.9.1) has undefined behavior when it tries to decode a field that has another field-type then what is expected. Theoretically this could be detected at the cost of some additional checking. Other implementation may perform this check and then either ignore the field, return a protocol exception, or perform a silent type cast.

A **union** is encoded exactly the same as a struct with the additional restriction that at most 1 field may be encoded.

An **exception** is encoded exactly the same as a struct.

3. Thrift Binary protocol encoding

3.1. Integer encoding

In the *binary protocol* integers are encoded with the most significant byte first (big endian byte order, aka network order). An `i8` needs 1 byte, an `i16` 2, an `i32` 4 and an `i64` needs 8 bytes.

The CPP version has the option to use the binary protocol with little endian order. Little endian gives a small but noticeable performance boost because contemporary CPUs use little endian when storing integers to RAM.

3.2. Enum encoding

The generated code encodes enums by taking the ordinal value and then encoding that as an i32.

3.3. Binary encoding

Binary is sent as follows:

```
Binary protocol, binary data, 4+ bytes:
+--------+--------+--------+--------+--------+...+--------+
| byte length                       | bytes               |
+--------+--------+--------+--------+--------+...+--------+
```

Where:

- `byte length` is the length of the byte array, a signed 32 bit integer encoded in network (big endian) order (must be >= 0).
- `bytes` are the bytes of the byte array.

Be default the length is limited to 2147483647, however some implementation have the option to lower the limit.

3.4. String encoding

Strings are first encoded to UTF-8, and then send as binary.

3.5. Double encoding

Values of type `double` are first converted to an i64 according to the IEEE 754 floating-point "double format" bit layout. Most run-times provide primitives for the conversion. The i64 is encoded using 8 bytes in big endian order.

This is some scala code showing the JVM primitives to convert from double to i64 and back:

```
def doubleToI64(d: Double): Long = java.lang.Double.doubleToLongBits(d)
def i64ToDouble(l: Long): Double = java.lang.Double.longBitsToDouble(l)
```

3.6. Boolean encoding

Values of `bool` type are first converted to an i8. True is converted to `1`, false to `0`.

3.7. Message encoding

A `Message` can be encoded in two different ways, the modern *strict encoding*, or the nameless old encoding.

```
Binary protocol Message, strict encoding, 12+ bytes:
+--------+--------+--------+--------+--------+--------+--------+--------+--------+...+--
|1vvvvvvv|vvvvvvvv|unused  |00000mmm| name length                       | name
+--------+--------+--------+--------+--------+--------+--------+--------+--------+...+--
```

Where:

- `vvvvvvvvvvvvvvv` is the version, an unsigned 15 bit number fixed to `1` (in binary: `000 0000 0000 0001`). The leading bit is `1`.

- `unused` is an ignored byte.
- `mmm` is the message type, an unsigned 3 bit integer. The 5 leading bits must be `0` as some clients (checked for java in 0.9.1) take the whole byte.
- `name length` is the byte length of the name field, a signed 32 bit integer encoded in network (big endian) order (must be >= 0).
- `name` is the method name, a UTF-8 encoded string.
- `seq id` is the sequence id, a signed 32 bit integer encoded in network (big endian) order.

The second, older encoding (aka non-strict) is:

```
Binary protocol Message, old encoding, 9+ bytes:
+--------+--------+--------+--------+--------+...+--------+--------+--------+--------+--
| name length                       | name                |00000mmm| seq id
+--------+--------+--------+--------+--------+...+--------+--------+--------+--------+--
```

Where `name length`, `name`, `mmm`, `seq id` are as above.

Because `name length` must be positive (therefore the first bit is always `0`), the first bit allows the receiver to see whether the strict format or the old format is used. Therefore a server and client using the different variants of the binary protocol can transparently talk with each other. However, when strict mode is enforced, the old format is rejected.

Message types are encoded with the following values:

- *Call*: 1
- *Reply*: 2
- *Exception*: 3
- *Oneway*: 4

3.8. Struct encoding

In the binary protocol field headers and the stop field are encoded as follows:

```
Binary protocol field header and field value:
+--------+--------+--------+--------+...+--------+
|tttttttt| field id        | field value         |
+--------+--------+--------+--------+...+--------+

Binary protocol stop field:
+--------+
|00000000|
+--------+
```

Where:

- `tttttttt` the field-type, a signed 8 bit integer.
- `field id` the field-id, a signed 16 bit integer in big endian order.

- `field-value` the encoded field value.

The following field-types are used:

- `bool`, encoded as `2`
- `byte`, encoded as `3`
- `double`, encoded as `4`
- `i16`, encoded as `6`
- `i32`, encoded as `8`
- `i64`, encoded as `10`
- `string`, used for binary and string fields, encoded as `11`
- `struct`, used for structs and union fields, encoded as `12`
- `map`, encoded as `13`
- `set`, encoded as `14`
- `list`, encoded as `15`

3.9. List and Set

List and sets are encoded the same: a header indicating the size and the element-type of the elements, followed by the encoded elements.

```
Binary protocol list (5+ bytes) and elements:
+--------+--------+--------+--------+--------+--------+...+--------+
|tttttttt| size                              | elements            |
+--------+--------+--------+--------+--------+--------+...+--------+
```

Where:

- `tttttttt` is the element-type, encoded as an i8
- `size` is the size, encoded as an i32, positive values only
- `elements` the element values

The element-type values are the same as field-types. The full list is included in the struct section above.

The maximum list/set size is configurable. By default there is no limit (meaning the limit is the maximum i32 value: 2147483647).

3.10. Map

Maps are encoded with a header indicating the size, the element-type of the keys and the element-type of the elements, followed by the encoded elements. The encoding follows this BNF:

```
map  ::=  key-element-type value-element-type size ( key value )*

Binary protocol map (6+ bytes) and key value pairs:
+--------+--------+--------+--------+--------+--------+--------+...+--------+
|kkkkkkkk|vvvvvvvv| size                                | key value pairs     |
+--------+--------+--------+--------+--------+--------+--------+...+--------+
```

Where:

- `kkkkkkkk` is the key element-type, encoded as an i8
- `vvvvvvvv` is the value element-type, encoded as an i8
- `size` is the size of the map, encoded as an i32, positive values only
- `key value pairs` are the encoded keys and values

The element-type values are the same as field-types. The full list is included in the struct section above.

The maximum map size is configurable. By default there is no limit (meaning the limit is the maximum i32 value: 2147483647).

4. Thrift compact protocol encoding

4.1. Integer encoding

The *compact protocol* uses multiple encodings for integers: the *zigzag int*, and the *var int*.

Values of type `i32` and `i64` are first transformed to a **zigzag int**. A zigzag int folds positive and negative numbers into the positive number space. When we read 0, 1, 2, 3, 4 or 5 from the wire, this is translated to 0, -1, 1, -2 or 2 respectively. Here are the (Scala) formulas to convert from i32/i64 to a zigzag int and back:

```
def i32ToZigZag(n: Int): Int = (n << 1) ^ (n >> 31)
def zigzagToInt32(n: Int): Int = (n >>> 1) ^ - (n & 1)
def i64ToZigZag(n: Long): Long = (n << 1) ^ (n >> 63)
def zigzagToI64(n: Long): Long = (n >>> 1) ^ - (n & 1)
```

The zigzag int is then encoded as a **var int**. Var ints take 1 to 5 bytes (i32) or 1 to 10 bytes (i64). The most significant bit of each byte indicates if more bytes follow. The concatenation of the least significant 7 bits from each byte form the number, where the first byte has the most significant bits (so they are in big endian or network order).

Var ints are sometimes used directly inside the compact protocol to represent numbers that are usually positive.

To encode an `i16` as zigzag int, it is first converted to an `i32` and then encoded as such. The type `i8` simply uses a single byte as in the binary protocol.

4.2. Enum encoding

The generated code encodes enums by taking the ordinal value and then encoding that just like an i32.

4.3. Binary encoding

Binary is sent as follows:

```
Binary protocol, binary data, 1+ bytes:
+--------+...+--------+--------+...+--------+
| byte length         | bytes                |
+--------+...+--------+--------+...+--------+
```

Where:

- `byte length` is the length of the byte array, using var int encoding (must be >= 0).
- `bytes` are the bytes of the byte array.

Be default the length is limited to 2147483647, however some implementation have the option to lower the limit.

4.4. String encoding

Strings are first encoded to UTF-8, and then send as binary.

4.5. Double encoding

Values of type `double` are first converted to an i64 according to the IEEE 754 floating-point "double format" bit layout. Most run-times provide primitives for the conversion. The i64 is encoded using 8 bytes in big endian order.

This is some scala code showing the JVM primitives to convert from double to i64 and back:

```
def doubleToI64(d: Double): Long = java.lang.Double.doubleToLongBits(d)
def i64ToDouble(l: Long): Double = java.lang.Double.longBitsToDouble(l)
```

4.6. Boolean encoding

Booleans are encoded differently depending on whether it is a field value (in a struct) or an element value (in a set, list or map). Field values are encoded directly in the field header. Element values of type bool are sent as an i8; true as `1` and false as `0`.

4.7. Message encoding

A `Message` on the wire looks as follows:

```
Compact protocol Message (4+ bytes):
+--------+--------+--------+...+--------+--------+...+--------+--------+...+--------+
|pppppppp|mmmvvvvv| seq id              | name length         | name                |
+--------+--------+--------+...+--------+--------+...+--------+--------+...+--------+
```

Where:

- `pppppppp` is the protocol id, fixed to `1000 0010` or `0x82`.

- `mmm` is the message type, an unsigned 3 bit integer.
- `vvvvv` is the version, an unsigned 5 bit integer, fixed to `00001`.
- `seq id` is the sequence id, a signed 32 bit integer encoded as a var int.
- `name length` is the byte length of the name field, a signed 32 bit integer encoded as a var int (must be >= 0).
- `name` is the method name to invoke, a UTF-8 encoded string.

Message types are encoded with the following values:

- *Call*: 1
- *Reply*: 2
- *Exception*: 3
- *Oneway*: 4

4.8. Struct encoding

```
Compact protocol field header (short form) and field value:
+--------+--------+...+--------+
|ddddtttt| field value         |
+--------+--------+...+--------+

Compact protocol field header (1 to 3 bytes, long form) and field value:
+--------+--------+...+--------+--------+...+--------+
|0000tttt| field id            | field value         |
+--------+--------+...+--------+--------+...+--------+

Compact protocol stop field:
+--------+
|00000000|
+--------+
```

Where:

- `dddd` is the field id delta, an unsigned 4 bits integer (strictly positive, e.g. > 0).
- `tttt` is field-type id, an unsigned 4 bit integer.
- `field id` the field id, a signed 16 bit integer encoded as zigzag int.
- `field-value` the encoded field value.

The field id delta can be computed by `current-field-id - previous-field-id`, or just `current-field-id` if this is the first of the struct. The short form *should* be used when the field id delta is in the range 1 - 15 (inclusive).

The following field-types/values can be encoded:

- `bool` with value `true`, encoded as `1`

- `bool` with value `false`, encoded as `2`
- `byte`, encoded as `3`
- `i16`, encoded as `4`
- `i32`, encoded as `5`
- `i64`, encoded as `6`
- `double`, encoded as `7`
- `binary`, used for binary and string fields, encoded as `8`
- `list`, encoded as `9`
- `set`, encoded as `10`
- `map`, encoded as `11`
- `struct`, used for both structs and union fields, encoded as `12`

Note that because there are 2 specific field types for the boolean values, the encoding of a boolean field value has no length (0 bytes).

4.9. List and Set

List and sets are encoded the same: a header indicating the size and the element-type of the elements, followed by the encoded elements.

```
Compact protocol list header (1 byte, short form) and elements:
+--------+--------+...+--------+
|sssstttt| elements            |
+--------+--------+...+--------+

Compact protocol list header (2+ bytes, long form) and elements:
+--------+--------+...+--------+--------+...+--------+
|1111tttt| size               | elements            |
+--------+--------+...+--------+--------+...+--------+
```

Where:

- `ssss` is the size, 4 bit unsigned integer, values `0` - `14`
- `tttt` is the element-type, a 4 bit unsigned integer
- `size` is the size, a var int (i32), positive values `15` or higher
- `elements` are the encoded elements

The short form *should* be used when the length is in the range 0 - 14 (inclusive).

The following element-types are used (note that these are *different* from the field-types):

- `bool`, encoded as `2`

- `byte`, encoded as `3`
- `double`, encoded as `4`
- `i16`, encoded as `6`
- `i32`, encoded as `8`
- `i64`, encoded as `10`
- `string`, used for binary and string fields, encoded as `11`
- `struct`, used for structs and union fields, encoded as `12`
- `map`, encoded as `13`
- `set`, encoded as `14`
- `list`, encoded as `15`

The maximum list/set size is configurable. By default there is no limit (meaning the limit is the maximum i32 value: 2147483647).

4.10. Map

Maps are encoded with a header indicating the size, the type of the keys and the element-type of the elements, followed by the encoded elements. The encoding follows this BNF:

```
map           ::= empty-map | non-empty-map
empty-map     ::= `0`
non-empty-map ::= size key-element-type value-element-type (key value)+
```

```
Compact protocol map header (1 byte, empty map):
+--------+
|00000000|
+--------+
```

```
Compact protocol map header (2+ bytes, non empty map) and key value pairs:
+--------+...+--------+--------+--------+...+--------+
| size               |kkkkvvvv| key value pairs     |
+--------+...+--------+--------+--------+...+--------+
```

Where:

- `size` is the size, a var int (i32), strictly positive values (size > 0)
- `kkkk` is the key element-type, a 4 bit unsigned integer
- `vvvv` is the value element-type, a 4 bit unsigned integer
- `key value pairs` are the encoded keys and values

The element-types are the same as for lists. The full list is included in the *List and set* section.

The maximum map size is configurable. By default there is no limit (meaning the limit is the maximum i32 value: 2147483647).

5. Comparing binary and compact protocol

The binary protocol is fairly simple and therefore easy to process. The compact protocol needs less bytes to send the same data at the cost of additional processing. When bandwidth is a bottleneck, the compact protocol will be slightly faster. When bandwidth is not a concern there is no advantage.

Compatibility. A server could automatically determine whether a client talks the binary protocol or the compact protocol by investigating the first byte. If the value is `1000 0001` or `0000 0000` (assuming a name shorter then ±16 MB) it is the binary protocol. When the value is `1000 0010` it is talking the compact protocol.

6. Framed vs. unframed transport

The first thrift binary wire format was unframed. This means that information is sent out in a single stream of bytes. With unframed transport the (generated) processors will read directly from the socket (though Apache Thrift does try to grab all available bytes from the socket in a buffer when it can).

Later, Thrift introduced the framed transport.

With framed transport the full request and response (the message and the following struct) are first written to a buffer. Then when the struct is complete (transport method `flush` is hijacked for this), the length of the buffer is written to the socket first, followed by the buffered bytes. The combination is called a *frame*. On the receiver side the complete frame is first read in a buffer before the message is passed to a processor.

The length prefix is a 4 byte signed integer, send in network (big endian) order. The following must be true: `0` $\leq$ length $\leq$ `16384000` (16M).

Framed transport was introduced to ease the implementation of async processors. An async processor is only invoked when all data is received. Unfortunately, framed transport is not ideal for large messages as the entire frame stays in memory until the message has been processed. In addition, the java implementation merges the incoming data to a single, growing byte array. Every time the byte array is full it needs to be copied to a new larger byte array.

Framed and unframed transports are not compatible with each other.

7. BNF notation used in this document

The following BNF notation is used:

- a plus `+` appended to an item represents repetition; the item is repeated 1 or more times
- a star `*` appended to an item represents optional repetition; the item is repeated 0 or more times
- a pipe `|` between items represents choice, the first matching item is selected
- parenthesis `(` and `)` are used for grouping multiple items