Thrift: The Missing Guide

Diwaker Gupta <me@diwakergupta.info>

Revision History
2011-03-18 DG
Written against Thrift 0.6.0

Table of Contents

From the Thrift website [http://thrift.apache.org]:

> Thrift is a software framework for scalable cross-language services development. It combines a software stack with a code generation engine to build services that work efficiently and seamlessly between C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, and OCaml.

Thrift is clearly abundant in features. What is sorely lacking though is *good* documentation. This guide is an attempt to fill that hole. But note that this is a reference guide — for a step-by-step example on how to use Thrift, refer to the Thrift tutorial.

Many aspects of the structure and organization of this guide have been borrowed from the (excellent) Google Protocol Buffer Language Guide [http://code.google.com/apis/protocolbuffers/docs/proto.html]. I thank the authors of that document.

A PDF version [thrift.pdf] is also available.

Copyright. Copyright © 2011 Diwaker Gupta

This work is licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License [http://creativecommons.org/licenses/by-nc/3.0/].

Contributions. I welcome feedback and contributions to this guide. You can find the source code [https://github.com/diwakergupta/thrift-missing-guide] over at GitHub [http://github.com]. Alternatively, you can file a bug [https://github.com/diwakergupta/thrift-missing-guide/issues].

Acknowledgements. I thank the authors of Thrift for the software, the authors of the Google Protocol Buffer documentation for the inspiration and the Thrift community for the feedback. Special thanks to Dave Engberg from Evernote for his input.

About the Author. I'm an open source geek and a software architect. I blog over at Floating Sun [http://floatingsun.net] and you can find me here [http://diwakergupta.info].

1. Language Reference

1.1. Types

The Thrift type system consists of pre-defined base types, user-defined structs, container types, exceptions and service definitions.

Base Types

- `bool`: A boolean value (true or false), one byte
- `byte`: A signed byte
- `i16`: A 16-bit signed integer
- `i32`: A 32-bit signed integer
- `i64`: A 64-bit signed integer
- `double`: A 64-bit floating point number
- `string`: Encoding agnostic text or binary string

Note that Thrift does not support unsigned integers because they have no direct translation to native (primitive) types in many of Thrift's target languages.

Containers

Thrift containers are strongly typed containers that map to the most commonly used containers in popular programming languages. They are annotated using the Java Generics style. There are three containers types available:

- `list<t1>`: An ordered list of elements of type `t1`. May contain duplicates.
- `set<t1>`: An unordered set of unique elements of type `t1`.
- `map<t1,t2>`: A map of strictly unique keys of type `t1` to values of type `t2`.

Types used in containers many be any valid Thrift type (including structs and exceptions) excluding services.

Structs and Exceptions

A Thrift struct is conceptually similar to a `C` struct — a convenient way of grouping together (and encapsulating) related items. Structs translate to classes in object-oriented languages.

Exceptions are syntactically and functionally equivalent to structs except that they are declared using the `exception` keyword instead of the `struct` keyword. They differ from structs in semantics — when defining RPC services, developers may declare that a remote method throws an exception.

Details on defining structs and exceptions are the subject of a later section.

Services

Service definitions are semantically equivalent to defining an `interface` (or a pure virtual abstract class) in object-oriented programming. The Thrift compiler generates fully functional client and server stubs that implement the interface.

Details on defining services are the subject of a later section.

1.2. Typedefs

Thrift supports C/C++ style typedefs.

```
typedef i32 MyInteger   // 1
typedef Tweet ReTweet   // 2
```

1 Note there is no trailing semi-colon
2 Structs can also be used in typedefs

1.3. Enums

When you're defining a message type, you might want one of its fields to only have one of a pre-defined list of values. For example, let's say you want to add a `tweetType` field for each `Tweet`, where the `tweetType` can be `TWEET`, `RETWEET`, `DM`, or `REPLY`. You can do this very simply by adding an enum to your message definition — a field with an enum type can only have one of a specified set of constants as its value (if you try to provide a different value, the parser will treat it like an unknown field). In the following example we've added an enum called `TweetType` with all the possible values, and a field of the same type:

```
enum TweetType {
    TWEET,       // 1
    RETWEET = 2, // 2
    DM = 0xa,    // 3
    REPLY
}                // 4

struct Tweet {
    1: required i32 userId;
    2: required string userName;
    3: required string text;
    4: optional Location loc;
```

```
    5: optional TweetType tweetType = TweetType.TWEET // 5
    16: optional string language = "english"
}
```

1 Enums are specified C-style. Compiler assigns default values starting at 0.
2 You can of course, supply specific integral values for constants.
3 Hex values are also acceptable.
4 Again notice no trailing semi-colon
5 Use the fully qualified name of the constant when assigning default values.

Note that unlike Protocol Buffers, Thrift does NOT yet support nested enums (or structs, for that matter).

Enumerator constants must be in the range of *postive* 32-bit integers.

1.4. Comments

Thrift supports shell-style, C-style multi-line as well as single-line Java/C++ style comments.

```
# This is a valid comment.

/*
 * This is a multi-line comment.
 * Just like in C.
 */

// C++/Java style single-line comments work just as well.
```

1.5. Namespaces

Namespaces in Thrift are akin to namespaces in C++ or packages in Java — they offer a convenient way of organizing (or isolating) your code. Namespaces may also be used to prevent name clashes between type definitions.

Because each language has its own package-like mechanisms (e.g. Python has modules), Thrift allows you to customize the namespace behavior on a per-language basis:

```
namespace cpp com.example.project  // 1
namespace java com.example.project // 2
```

1 Translates to `namespace com { namespace example { namespace project {`
2 Translates to `package com.example.project`

1.6. Includes

It is often useful to split up Thrift definitions in separate files to ease maintainance, enable reuse and improve modularity/organization. Thrift allows files to *include* other Thrift files. Included files are looked up in the current directory and by searching relative to any paths specified with the `-I` compiler flag.

Included objects are accessed using the name of the Thrift file as a prefix.

```
include "tweet.thrift"           // 1
```

```
...
struct TweetSearchResult {
    1: list<tweet.Tweet> tweets; // 2
}
```

1 File names must be quoted; again notice the absent semi-colon.
2 Note the `tweet` prefix.

1.7. Constants

Thrift lets you define constants for use across languages. Complex types and structs are specified using JSON notation.

```
const i32 INT_CONST = 1234;    // 1
const map<string,string> MAP_CONST = {"hello": "world", "goodnight": "moon"}
```

1 Semi-colon is (confusingly) optional; hex values are valid here.

1.8. Defining Structs

Structs (also known as *messages* in some systems) are the basic building blocks in a Thrift IDL. A struct is composed of *fields*; each field has a unique integer identifier, a type, a name and an optional default value.

Consider a simple example. Suppose you want to build a Twitter [http://twitter.com]-like service. Here is how may define a `Tweet`:

```
struct Tweet {
    1: required i32 userId;                  // 1
    2: required string userName;             // 2
    3: required string text;
    4: optional Location loc;                // 3
    16: optional string language = "english" // 4
}

struct Location {                            // 5
    1: required double latitude;
    2: required double longitude;
}
```

1 Every field **must** have a unique, positive integer identifier
2 Fields may be marked as `required` or `optional`
3 Structs may contain other structs
4 You may specify an optional "default" value for a field
5 Multiple structs can be defined and referred to within the same Thrift file

As you can see, each field in the message definition has a unique numbered tag. These tags are used to identify your fields in the wire format, and should not be changed once your message type is in use.

Fields may be marked `required` or `optional` with obvious meanings for well-formed structs. Thrift will complain if required fields have not been set in a struct, for instance. If an optional field has not been set in the struct, it will not be serialized over the wire. If a default value has been specified for an optional field, the field is assigned the default value when the struct is parsed and no value has been explicitly assigned for that field.

Unlike services, structs do not support inheritance, that is, a struct may not extend other structs.

Warning

You should be very careful about marking fields as required. If at some point you wish to stop writing or sending a required field, it will be problematic to change the field to an optional field — old readers will consider messages without this field to be incomplete and may reject or drop them unintentionally. You should consider writing application-specific custom validation routines for your buffers instead. Some have come the conclusion that using required does more harm than good; they prefer to use only optional and repeated. However, this view is not universal.

1.9. Defining Services

While there are several popular serialization/deserialization frameworks (like Protocol Buffers), there are few frameworks that provide out-of-the-box support for RPC-based services across multiple languages. This is one of the major attractions of Thrift.

Think of service definitions as Java interfaces — you need to supply a name and signatures for the methods. Optionally, a service may extend other services.

The Thrift compiler will generate service interface code (for the server) and stubs (for the client) in your chosen language. Thrift ships with RPC libraries for most languages that you can then use to run your client and server.

```
service Twitter {
    // A method definition looks like C code. It has a return type, arguments,
    // and optionally a list of exceptions that it may throw. Note that argument
    // lists and exception list are specified using the exact same syntax as
    // field lists in structs.
    void ping(),                                    // 1
    bool postTweet(1:Tweet tweet);                  // 2
    TweetSearchResult searchTweets(1:string query); // 3

    // The 'oneway' modifier indicates that the client only makes a request and
    // does not wait for any response at all. Oneway methods MUST be void.
    oneway void zip()                               // 4
}
```

1. Confusingly, method definitions can be terminated using comma or semi-colon
2. Arguments can be primitive types or structs
3. Likewise for return types
4. `void` is a valid return type for functions

Note that the argument lists (and exception lists) for functions are specified exactly like structs.

Services support inheritance: a service may optionally inherit from another service using the `extends` keyword.

Important

As of this writing, Thrift does NOT nested type *definitions*. That is, you may not define a struct (or an enum) within a struct; you may of course *use* structs/enums within other structs.

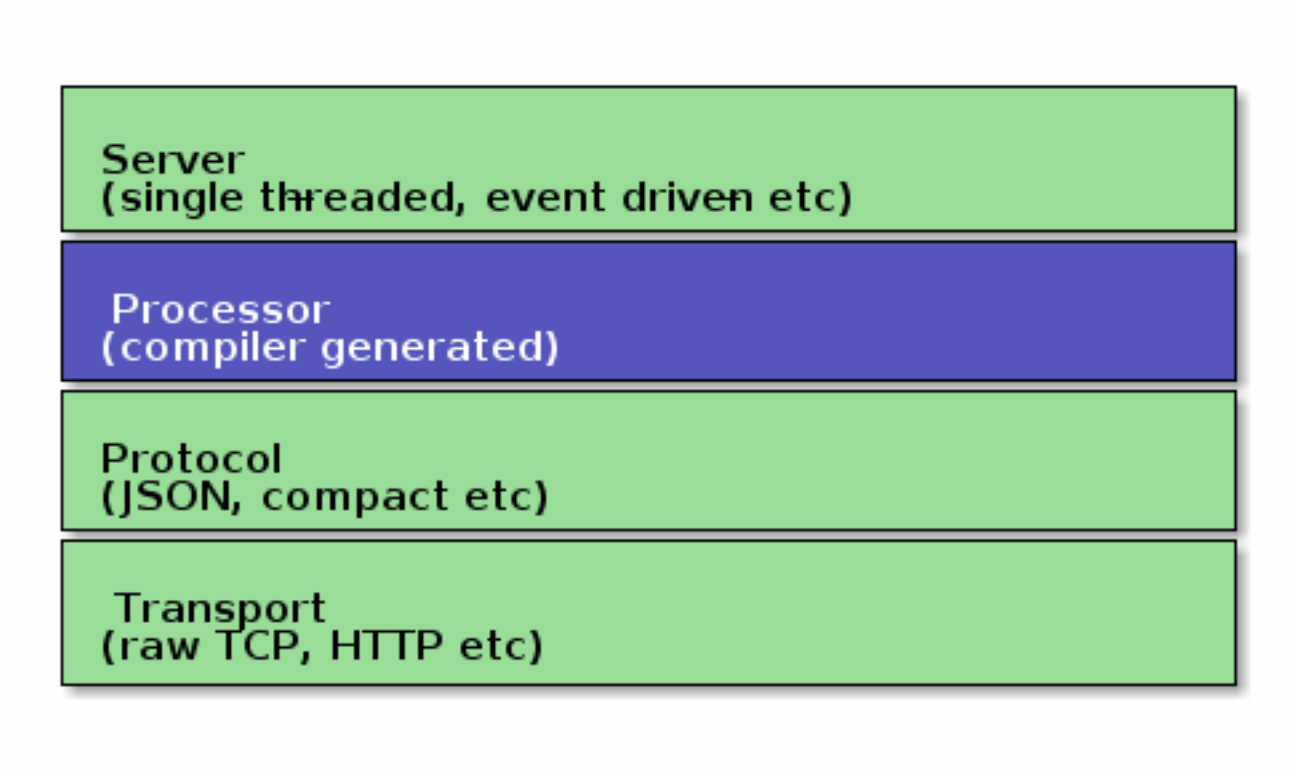
2. Generated Code

This section contains documentation for working with Thrift generated code in various target languages. We begin by introducing the common concepts that are used across the board — these govern how the generated code is structured and will hopefully help you understand how to use it effectively.

2.1. Concepts

Here is a pictorial view of the Thrift network stack:

Figure 1. The Thrift Network Stack



Transport

The Transport layer provides a simple abstraction for reading/writing from/to the network. This enables Thrift to decouple the underlying transport from the rest of the system (serialization/ deserialization, for instance).

Here are some of the methods exposed by the `Transport` interface:

- `open`
- `close`
- `read`
- `write`

- `flush`

In addition to the `Transport` interface above, Thrift also uses a `ServerTransport` interface used to accept or create primitive transport objects. As the name suggest, `ServerTransport` is used mainly on the server side to create new Transport objects for incoming connections.

- `open`
- `listen`
- `accept`
- `close`

Protocol

The Protocol abstraction defines a mechanism to map in-memory data structures to a wire-format. In other words, a protocol specifies how datatypes use the underlying Transport to encode/decode themselves. Thus the protocol implementation governs the encoding scheme and is responsible for (de)serialization. Some examples of protocols in this sense include JSON, XML, plain text, compact binary etc.

Here is the `Protocol` interface:

```
writeMessageBegin(name, type, seq)
writeMessageEnd()
writeStructBegin(name)
writeStructEnd()
writeFieldBegin(name, type, id)
writeFieldEnd()
writeFieldStop()
writeMapBegin(ktype, vtype, size)
writeMapEnd()
writeListBegin(etype, size)
writeListEnd()
writeSetBegin(etype, size)
writeSetEnd()
writeBool(bool)
writeByte(byte)
writeI16(i16)
writeI32(i32)
writeI64(i64)
writeDouble(double)
writeString(string)

name, type, seq = readMessageBegin()
                  readMessageEnd()
name = readStructBegin()
       readStructEnd()
name, type, id = readFieldBegin()
                 readFieldEnd()
k, v, size = readMapBegin()
             readMapEnd()
etype, size = readListBegin()
              readListEnd()
etype, size = readSetBegin()
              readSetEnd()
bool = readBool()
```

```
byte = readByte()
i16 = readI16()
i32 = readI32()
i64 = readI64()
double = readDouble()
string = readString()
```

Thrift Protocols are stream oriented by design. There is no need for any explicit framing. For instance, it is not necessary to know the length of a string or the number of items in a list before we start serializing them.

Processor

A Processor encapsulates the ability to read data from input streams and write to output streams. The input and output streams are represented by Protocol objects. The Processor interface is extremely simple:

```
interface TProcessor {
    bool process(TProtocol in, TProtocol out) throws TException
}
```

Service-specific processor implementations are generated by the compiler. The Processor essentially reads data from the wire (using the input protocol), delegates processing to the handler (implemented by the user) and writes the response over the wire (using the output protocol).

Server

A Server pulls together all of the various features described above:

- Create a transport
- Create input/output protocols for the transport
- Create a processor based on the input/output protocols
- Wait for incoming connections and hand them off to the processor

Next we discuss the generated code for specific languages. Unless mentioned otherwise, the sections below will assume the following Thrift specification:

Example IDL.

```
namespace cpp thrift.example
namespace java thrift.example

enum TweetType {
    TWEET,
    RETWEET = 2,
    DM = 0xa,
    REPLY
}

struct Location {
    1: required double latitude;
    2: required double longitude;
}
```

```
struct Tweet {
    1: required i32 userId;
    2: required string userName;
    3: required string text;
    4: optional Location loc;
    5: optional TweetType tweetType = TweetType.TWEET;
    16: optional string language = "english";
}

struct TweetSearchResult {
    1: list<Tweet> tweets;
}

const i32 MAX_RESULTS = 100;

service Twitter {
    void ping(),
    bool postTweet(1:Tweet tweet);
    TweetSearchResult searchTweets(1:string query);
    oneway void zip()
}
```

How are nested structs initialized?

In an earlier section, we saw how Thrift allows structs to contain other structs (no nested definitions yet though!) In most object-oriented and/or dynamic languages, structs map to objects and so it is instructive to understand how Thrift initializes nested structs. One reasonable approach would be to treat the nested structs as pointers or references and initialize them with NULL, until explicitly set by the user.

Unfortunately, for many languages, Thrift uses a *pass by value* model. As a concrete example, consider the generated C++ code for the `Tweet` struct in our example above:

```
...
int32_t userId;
std::string userName;
std::string text;
Location loc;
TweetType::type tweetType;
std::string language;
...
```

As you can see, the nested `Location` structure is **fully allocated inline**. Because `Location` is optional, the code uses the internal *__isset* flags to determine if the field has actually been "set" by the user.

This can lead to some surprising and unintuitive behavior:

- Since the full size of every sub-structure may be allocated at initialization in some languages, memory usage may be higher than you expect, especially for complicated structures with many unset fields.
- The parameters and return types for service methods may not be "optional" and you can’t assign or return `null` in any dynamic language. Thus to return a "no value" result from a method, you must declare an envelope structure with an optional field containing the value and then return the envelope with that field unset.
- The transport layer can, however, marshal method calls from older versions of a service definition with missing parameters. Thus, if the original service contained a method `postTweet(1: Tweet tweet)` and a later version changes it to `postTweet(1: Tweet tweet, 2: string group)`, then an older client invoking the previous method will result in a newer server receiving the call with the new parameter unset. If the new server is in Java, for instance, you may in fact receive a `null` value for the new parameter. And yet you may not declare a parameter to be nullable within the IDL.

2.2. Java

Generated Files

- a single file (`Constants.java`) containing all constant definitions
- one file per struct, enum and service

```
$ tree gen-java
`-- thrift
    `-- example
        |-- Constants.java
        |-- Location.java
        |-- Tweet.java
        |-- TweetSearchResult.java
        |-- TweetType.java
        `-- Twitter.java
```

Naming Conventions

While the Thrift compiler does not enforce any naming conventions, it is advisable to stick to standard naming conventions otherwise you man be in for some surprises. For instance, if you have a struct named `tweetSearchResults` (note the mixedCase), the Thrift compiler will generated a Java file named `TweetSearchResults` (note the CamelCase) containing a class named `tweetSearchResults` (like the original struct). This will obviously not compile under Java.

2.3. C++

Generated Files

- all constants go into a single `.cpp/.h` pair
- all type definitions (enums and structs) go into another `.cpp/.h` pair
- each service gets its own `.cpp/.h` pair

```
$ tree gen-cpp
|-- example_constants.cpp
|-- example_constants.h
|-- example_types.cpp
|-- example_types.h
|-- Twitter.cpp
|-- Twitter.h
`-- Twitter_server.skeleton.cpp
```

2.4. Other Languages

Python, Ruby, Javascript etc.

3. Best Practices

3.1. Versioning/Compatibility

Protocols evolve over time. If an existing message type no longer meets all your needs — for example, you'd like the message format to have an extra field — but you'd still like to use code created with the old format, don't worry! It's very simple to update message types without breaking any of your existing code. Just remember the following rules:

- Don’t change the numeric tags for any existing fields.
- Any new fields that you add should be optional. This means that any messages serialized by code using your "old" message format can be parsed by your new generated code, as they won’t be missing any required elements. You should set up sensible default values for these elements so that new code can properly interact with messages generated by old code. Similarly, messages created by your new code can be parsed by your old code: old binaries simply ignore the new field when parsing. However, the unknown fields are not discarded, and if the message is later serialized, the unknown fields are serialized along with it — so if the message is passed on to new code, the new fields are still available.
- Non-required fields can be removed, as long as the tag number is not used again in your updated message type (it may be better to rename the field instead, perhaps adding the prefix "OBSOLETE_", so that future users of your .thrift can’t accidentally reuse the number).
- Changing a default value is generally OK, as long as you remember that default values are never sent over the wire. Thus, if a program receives a message in which a particular field isn’t set, the program will see the default value as it was defined in that program’s version of the protocol. It will NOT see the default value that was defined in the sender’s code.

4. Resources

- Thrift whitepaper [http://thrift.apache.org/static/thrift-20070401.pdf]
- Thrift Tutorial [http://wiki.apache.org/thrift/Tutorial]
- Thrift Wiki [http://wiki.apache.org/thrift]
- Protocol Buffers [http://code.google.com/apis/protocolbuffers/docs/overview.html]