Protocol Buffer vs Thrift vs Avro
Basic questions are:
- What kind of protocol to use, and what data to transmit?
- Efficient mechanism for storing and exchanging data
- What to do with requests on the server side?
Why can’t we use any of the protocols???

- SOAP
- CORBA
- COM, DCOM+
- JSON, Plain Text, XML
Should we pick any one of these? (NO)

- **SOAP**
  - XML, XML and more XML. Do we really need to parse so much XML?

- **CORBA**
  - Amazing idea, horrible execution
  - Overdesigned and heavyweight

- **DCOM, COM+**
  - Embraced mainly in windows client software

- **HTTP/JSON/XML/Plain Text**
  - Okay, proven – hurray!
  - But lack protocol description.
  - You have to maintain both client and server code.
  - XML has high parsing overhead.
  - (relatively) expensive to process; large due to repeated tags
Serialization Frameworks

XML, JSON,

Protocol Buffers, BERT,

BSON, Apache Thrift, Message Pack,

kryo, Apache Avro,

Custom Protocol...
Common Properties in Serialization Frameworks

• Interface Description (IDL)
• Performance
• Versioning
• Binary Format
Google Protobuff

• Designed ~2001 because everything else wasn’t that good those days
• Production, proprietary in Google from 2001-2008, open-sourced since 2008
• Battle tested, very stable, well trusted
• Every time you hit a Google page, you're hitting several services and several PB code
• PB is the glue to all Google services
• Official support for four languages: C++, Java, Python, and JavaScript
• Does have a lot of third-party support for other languages (of highly variable quality)
• Current Version - protobuf-2.5.0
• BSD License
Apache THRIFT

- Designed by an X-Googler in 2007
- Developed internally at Facebook, used extensively there
- An open Apache project, hosted in Apache's Inkubator.
- Aims to be the next-generation PB (e.g. more comprehensive features, more languages)
- IDL syntax is slightly cleaner than PB. If you know one, then you know the other
- Supports: C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi and other languages
- Offers a stack for RPC calls
- Current Version - thrift-0.9.0
- Apache License 2.0
Typical Operation Model

- The typical model of Thrift/Protobuf use is
  - Write down a bunch of struct-like message formats in an IDL-like language.
  - Run a tool to generate Java/C++/whatever boilerplate code.
    - Example: `thrift --gen java MyProject.thrift`
  - Outputs thousands of lines - but they remain fairly readable in most languages
  - Link against this boilerplate when you build your application.
  - **DO NOT EDIT!**
Create a thrift file eg demo.thrift

Thrift Code Generator Tool (written in C++)

Define Data types and Service interfaces

Build Thrift platform files

Demo.php  Demo.cpp  Demo.py  Demo.java

Create Server/Client App Run the Server

Server implements Services and Client calls them
Interface Definition Language (IDL)

- **IDL** is a specification language used to describe a software component's interface.
- IDLs describe an interface in a language-independent way, enabling communication between software components that do not share a language – for example, between components written in C++ and components written in Java.
- IDLs are commonly used in remote procedure call software.
Thrift

```plaintext	namespace java serializers.thrift.media

typedef i32 int
typedef i64 long

enum Size {
    SMALL = 0,
    LARGE = 1,
}
enum Player {
    JAVA = 0,
    FLASH = 1,
}

struct Image {
    1: string uri, //url to the images
    2: optional string title,
    3: required int width,
    4: required int height,
    5: required Size size,
}

struct Media {
    1: string uri, //url to the thumbnail
    2: optional string title,
    3: required int width,
    4: required int height,
    5: required list<string> person,
    6: required Player player,
    7: optional string copyright,
}

struct MediaContent {
    1: required list<Image> image,
    2: required Media media,
}
```

Protobuf

```plaintext
package serializers.protobuf.media;

option java_package = "serializers.protobuf.media";
option java_outer_classname = "MediaContentHolder";
option optimize_for = SPEED; affects the C++ and Java code generators

message Image {
    required string uri = 1; //url to the thumbnail
    optional string title = 2; //used in the html
    required int32 width = 3; // of the image
    required int32 height = 4; // of the image
    enum Size {
        SMALL = 0;
        LARGE = 1;
    }
    required Size size = 5;
}

message Media {
    required string uri = 1;
    optional string title = 2;
    required int32 width = 3;
    required int32 height = 4;
    repeated string person = 5;
    enum Player {
        JAVA = 0;
        FLASH = 1;
    }
    required Player player = 6;
    optional string copyright = 7;
}

message MediaContent {
    repeated Image image = 1;
    required Media media = 2;
}
```
Defining IDL Rules

• Every field **must** have a unique, positive integer identifier ("= 1", " = 2" or " 1:", " 2:" )
• Fields may be marked as ’required’ or ’optional’
• structs/messages may contain other structs/messages
• You may specify an optional "default" value for a field
• Multiple structs/messages can be defined and referred to within the same .thrift/.proto file
Java Example (Person.proto)

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}

Person.PhoneNumber.newBuilder()
enum PhoneType {
  MOBILE = 0;
  HOME = 1;
  WORK = 2;
}

message PhoneNumber {
  required string number = 1;
  optional PhoneType type = 2 [default = HOME];
}

repeated PhoneNumber phone = 4;

Person john =
  Person.newBuilder()
    .setId(1234)
    .setEmail("jdoe@example.com")
    .setName("John Doe")
    .addPhone()
    .setNumber("555-4321")
    .setType(Person.PhoneType.HOME)
    .build();
<table>
<thead>
<tr>
<th></th>
<th>Thrift</th>
<th>Protocol Buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Composite Type</strong></td>
<td>Struct {}</td>
<td>Message {}</td>
</tr>
<tr>
<td><strong>Base Types</strong></td>
<td>bool</td>
<td>bool</td>
</tr>
<tr>
<td></td>
<td>byte</td>
<td>32/64-bit integers</td>
</tr>
<tr>
<td></td>
<td>16/32/64-bit integers</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td></td>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>byte sequence</td>
</tr>
<tr>
<td><strong>Containers</strong></td>
<td>list&lt;\texttt{t1}&gt;: An ordered list of elements of type \texttt{t1}. May contain duplicates.</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>\texttt{set&lt;\texttt{t1}&gt;}: An unordered set of unique elements of type \texttt{t1}.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\texttt{map&lt;\texttt{t1,\texttt{t2}&gt;}: A map of strictly unique keys of type \texttt{t1} to values of type \texttt{t2}.</td>
<td></td>
</tr>
<tr>
<td><strong>Enumerations</strong></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Constants</strong></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>\texttt{Example}:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const i32 INT_CONST = 1234;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const map&lt;\texttt{string,\texttt{string} &gt; MAP_CONST = {&quot;hello&quot;: &quot;world&quot;, &quot;goodnight&quot;: &quot;moon&quot;}</td>
<td></td>
</tr>
<tr>
<td><strong>Exception</strong></td>
<td>Yes (exception keyword instead of the struct keyword.)</td>
<td>No</td>
</tr>
<tr>
<td><strong>Type/Handling</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thrift</td>
<td>Protocol Buffers</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------</td>
<td>------------------</td>
</tr>
<tr>
<td>License</td>
<td>Apache</td>
<td>BSD-style</td>
</tr>
<tr>
<td>Compiler</td>
<td>C++</td>
<td>C++</td>
</tr>
<tr>
<td>RPC Interfaces</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>RPC Implementation</td>
<td>Yes</td>
<td>No (they do have one internally)</td>
</tr>
<tr>
<td>Composite Type Extensions</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Data Versioning</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Size Comparison

- Each write includes one Course object with 5 Person objects, and one Phone object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Size (smaller is better)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thrift — TCompactProtocol</td>
<td>278 (not bad)</td>
</tr>
<tr>
<td>Thrift — TBinaryProtocol</td>
<td>460</td>
</tr>
<tr>
<td>Protocol Buffers</td>
<td>250 (winner!)</td>
</tr>
<tr>
<td>RMI</td>
<td>905</td>
</tr>
<tr>
<td>REST — JSON</td>
<td>559</td>
</tr>
<tr>
<td>REST — XML</td>
<td>836</td>
</tr>
</tbody>
</table>

**TBinaryProtocol** – not optimized for space efficiency. Faster to process than the text protocol but more difficult to debug.

**TCompactProtocol** – More compact binary format; typically more efficient to process as well.
Versioning

- The system must be able to support reading of old data, as well as requests from out-of-date clients to new servers, and vice versa.
- Versioning in Thrift and Protobuf is implemented via field identifiers.
- The combination of this field identifiers and its type specifier is used to uniquely identify the field.
- A new compiling isn't necessary.
- Statically typed systems like CORBA or RMI would require an update of all clients in this case.
Projects using Thrift

Applications, projects, and organizations using Thrift include:

- Facebook
- Cassandra project
- Hadoop supports access to its HDFS API through Thrift bindings
- HBase leverages Thrift for a cross-language API
- Hypertable leverages Thrift for a cross-language API since v0.9.1.0a
- LastFM
- DoAT
- ThriftDB
- Scribe
- Evernote uses Thrift for its public API.
- Junkdepot
Projects using Protobuf

- Google
- ActiveMQ uses the protobuf for Message store
- Netty ([protobuf-rpc](#))
## Pros & Cons

<table>
<thead>
<tr>
<th>Pros</th>
<th>Thrift</th>
<th>Protocol Buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>More languages supported out of the box</td>
<td>Slightly faster than Thrift when using &quot;optimize_for = SPEED&quot;</td>
</tr>
<tr>
<td></td>
<td>Richer data structures than Protobuf (e.g.: Map and Set)</td>
<td>Serialized objects slightly smaller than Thrift due to more aggressive data compression</td>
</tr>
<tr>
<td></td>
<td>Includes RPC implementation for services</td>
<td>Better documentation</td>
</tr>
<tr>
<td></td>
<td>Good examples are hard to find</td>
<td>API a bit cleaner than Thrift</td>
</tr>
<tr>
<td></td>
<td>Missing/incomplete documentation</td>
<td>.proto can define services, but no RPC implementation is defined (although stubs are generated for you).</td>
</tr>
</tbody>
</table>
What about Avro?

- Avro is another very recent serialization system.
- Interoperability
  - Can Serialize into Avro/Binary or Avro/JSON
  - Supports reading and writing protobufs and thrift
- Supports multiple languages: Java, C, C++, C#, Python, Ruby
- Rich data structures with schema designed over JSON
  - A compact, fast, binary data format
  - A container file, to store persistent data (Schema ALWAYS Available)
  - RPC Framework
  - Schemas are equivalent to protocol buffers proto files, but they do not have to be generated.
- Simple integration with dynamic languages (via generic type)
  - Unlike other frameworks, unknown schema is not presented at runtime
- Compressible and Splitable by Hadoop MapReduce
Avro IDL Syntax [JSON]

Avro IDL:

```json
{
  "type": "record",
  "name": "BankDepositMsg",
  "fields": [
    {
      "name": "user_id",
      "type": "int"
    },
    {
      "name": "amount",
      "type": "double",
      "default": "0.00"
    },
    {
      "name": "datestamp",
      "type": "long"
    }
  ]
}
```

// Same Thrift IDL:

```thrift
struct BankDepositMsg {
  1: required i32 user_id;
  2: required double amount = 0.00;
  3: required i64 datestamp;
}
```
## Comparison with Thrift and PB

<table>
<thead>
<tr>
<th></th>
<th>Avro</th>
<th>Thrift and Protocol Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic schema</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Built into Hadoop</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Schema in JSON</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>No need to compile</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>No need to declare IDs</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Bleeding edge</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Sexy name 😊</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Comparison with other frameworks

- Avro provides functionality similar to systems such as Thrift, Protocol Buffers, etc.
- **Dynamic typing:** Avro does not require that code be generated. Data is always accompanied by a schema that permits full processing of that data without code generation, static datatypes, etc.
- **Untagged data:** Since the schema is present when data is read, considerably less type information need be encoded with data, resulting in smaller serialization size.
- **No manually-assigned field IDs:** When a schema changes, both the old and new schema are always present when processing data, so differences may be resolved symbolically, using field names.
Total Time ("total")
Create an object, serialize it to a byte array, then deserialize it back to an object.

Serialization Time ("ser")
Create an object, serialize it to a byte array.
- Java's built-in serializer faithfully represents arbitrary object graphs, which hurts performance. All the other serializers flatten the structure out to a tree.

Deserialization Time ("deser+deep")
Often the most expensive operation. To make a fair comparison, all fields of the deserialized instances are accessed - this forces lazy deserializers to really do their work. The raw data below shows additional measurements for deserialization.
The size of the serialized data. These numbers may vary depending on the exact data value being used.

- Java's built-in serializer stores the full class name in serialized form. So you don't need to know ahead of time what kind of object you're reading in.
- The `scala` test, which uses Java's built-in serialization, yields a larger serialized representation because it usually creates more Java classes under the hood.

### Serialization Compressed Size ("size+dll")

The size of the serialized data compressed with Java's built-in implementation of DEFLATE (zlib).
References:


http://www.slideshare.net/ChicagoHUG/avro-chug-20120416

http://www.slideshare.net/IgorAnishchenko/pb-vs-thrift-vs-avro