Encoding Data Formats for Evolvability

Arjun Reddy Lingala
arjunreddy.lingala@gmail.com

***Abstract*—Modern applications change over time, with more features modified or added as new products are launched and existing features are improved. Applications should aim to build systems that make it easy to adapt to changes. In most cases, a change to an application's features also requires a change in the data it stores, and a new field or record type must be captured or perhaps existing data must be presented in a new way. In order for the system to continue running smoothly, we need to maintain compatibility in both directions – backward compatibility and forward compatibility. Backward compatibility is normally not hard to achieve as an author of the newer code knows the format of data written by the older code, and so user can explicitly handle it (if necessary by simply keeping the old code to read the old data). Forward compatibility can be trickier because it requires older code to ignore additions made by a newer version of the code. In this paper, we will deep dive into formats for encoding data and look into how they handle schema changes and how they support systems where old and new data and code need to coexist. We will then discuss how those formats are used for data storage and for communication in web services, representational state transfer (REST) [2], and remote procedure calls (RPC) [3], as well as message-passing systems such as actors and message queues**

***Keywords*—Encoding, Decoding, Serialization, Deserialization, Forward Compatibility, Backward Compatibility, Evolvability, Performance, Compact**

I. Introduction

Modern applications usually store data in two types of representation, memory and disk. In memory, data are kept in objects, structs, lists, arrays, hash tables. These data structures are optimized for efficient access and manipulation by the CPU typically using pointers. When we want to write data to a file or send it over the network, we need to encode it as some kind of self-contained sequence of bytes, like JSON. Since the pointer does not make sense in this case, bytes representation in this case is different from the data structures used in memory. Hence, translation is required from in-memory representation to a byte sequence, which is called encoding or serialization. Many programming languages come with built-in support for encoding in-memory objects into byte sequences. For example, Java has *java.io.Serializable* [1], Ruby has *Marshal* [8], Python has *pickle* [9], and so on. Many third-party libraries also exist, such as *Kryo* for Java. Encoding libraries of programming languages are very appropriate as they allow in-memory objects to be saved and can be restored with minimal additional code. Though in-built programming languages have advantages, they also have few problems. The encoding is tied to programming language which will lead to reading in another programming language difficult. If we store or transmit data in such an encoding, we need to commit to current programming language for a very long time and building new applications in same programming language as they need communication. Versioning data is often an afterthought in these libraries: as they are intended for quick and easy encoding of data, they often neglect the inconvenient problems of forward and backward compatibility. In-built programming languages encoding formats are very bad at performance and provides more data through encoding like Java's serialization.

In next sections, we discuss various encoding formats starting from JSON, XML to Thrift [5], Protobuf [6] and Avro [4] and deep dive into how encoding works for these formats, how much data is generated by encoding using these formats, and how backward and forward compatibility works in these formats.

II. Standard Encoding

Standard encodings can be written and read by many programming languages and the obvious ones are JSON, XML and CSV in some cases. These encoding formats are widely known and are very easy to adapt, but are disliked and not used for being too verbose and complicated. JSON's popularity is mainly due to its built-in support in web browsers and simplicity relative to XML. JSON, XML, and CSV are textual formats, and thus human-readable, but has some subtle problems with these formats. There is a lot of ambiguity around the encoding of numbers. In XML and CSV, you cannot distinguish between a number and a string that happens to consist of digits. JSON distinguishes strings and numbers, but it does not distinguish integers and floating-point numbers, and it does not specify a precision. This is a problem when dealing with large numbers. For example, integers greater than 253 cannot be exactly represented in an IEEE 754 double-precision floating- point number, so such numbers become inaccurate when parsed in a language that uses floating-point numbers. This can be addressed by storing IDs twice, once as a JSON number and once as a decimal string, to work around the fact that the numbers are not correctly parsed by JavaScript applications. JSON and XML have good support for Unicode character strings, but they don't support binary strings. Binary strings are a useful feature, so people get around this limitation by encoding the binary data as text using Base64. The schema is then used to indicate that the value should be interpreted as Base64-encoded. CSV does not have any schema, so it is up to the application to define the meaning of each row and column. If an application change adds a new row or column, change needs to be handled manually. Although its escaping rules have been formally specified, not all parsers implement them correctly.

Despite these flaws, JSON, XML, and CSV are good enough for many purposes. It's likely that they will remain popular, especially as data interchange formats. In these situations, as long as people agree on what the format is, it often doesn't matter how pretty or efficient the format is.

III. Binary Encoding

Data that is used only internally within the organization, there is less pressure to use a common encoding format, an encoding format that is more compact or faster to parse can be used. Gains are negligible for small datasets, but once the data size gets into terabytes, data encoding format makes huge difference. JSON is less verbose than XML, but both still use a lot of space compared to binary formats. This observation led to the development of a binary encodings for JSON and for XML. These formats have been adopted in various niches, but none of them are as widely adopted as the textual versions of JSON and XML. Some of the binary formats of standard encoding formats extend the set of datatypes, but keeping the data model of JSON or XML unchanged. In particular, since they don't prescribe a schema, they need to include all the object field names within the encoded data. Lets take below example object and understand how the data is stored, how many bytes it takes for storage in various formats.

```
{
    "name": "Arjun",
    "id": 1234,
    "sports": ["cricket"]
}
```

MessagePack [7] is a binary encoding format of JSON and the above example object is encoded using MessagePack. At first it explains that an object is stored **0x80** and the number of fields the object contains **0x03**, combined together first byte contains 0x 83. The second byte, **0xa8**, indicates that what follows is a string. The next eight bytes are the field name userName in ASCII. Since the length was indicated previously. Following this approach, above object takes 34 bytes to store including the key names which is only little less than textual JSON encoding format with the cost of loss of human readability. In next sections we look into encoding formats Thrift, Protobuf approaches, same obeject can be stored in less number of bytes.

IV. Thrift and Protobuf

Apache Thrift [5] and Protocol Buffers (protobuf) [6] are binary encoding libraries that are based on the same principle. Protocol Buffers was originally developed at Google, Thrift was originally developed at Facebook, and both were made open source in 2007–08. Both Thrift and Protocol Buffers require a schema for any data that is encoded. Schema for above object in Thrift is defined in Thrift interface definition language as below:

```
struct Person {
    1: required string          name,
    2: required i64             id,
    3: optional list<string> sports
}
```

and the equivalent schema definition for Protocol Buffer looks as below:

```
message Person {
    required string name          = 1,
    required i64 id               = 2,
    optional list<string> sports  = 3
}
```

Thrift and Protocol Buffers each come with a code generation tool that takes a schema definition like the ones shown here, and produces classes that implement the schema in various programming languages which can be called from application code to encode or decode records of the schema. The main difference Thrift and Protobuf has compared to standard encoding data formats is that there are no field names stored with data as we have schema. Instead, the encoded data contains field tags, which are numbers (1, 2, and 3). Those are the numbers that appear in the schema definition. Field tags are like aliases for fields—they are a compact way of saying what field we are talking about, without having to spell out the field name. Same object which took 35 bytes for storage takes only 20 bytes of storage in Thirft protocol. It does this by packing the field type and tag number into a single byte, and by using variable-length integers. In thrift, for first field it take a byte for field tag and data type of the field, another byte for defining the length of the string and 5 bytes for the name field value. Next, field tag and data type takes a byte, integer value is stored in 2 bytes. Next, field tag and data type takes a byte, another byte to define the number of items in the list, a byte for the length of the value of the string and 7 bytes for the value of the field in the list. combining to a total of 20 bytes.

Protobuf does the encoding quite similar to Thrift protocol, but does the bit packing slightly differently for lists with providing field tag multiple times instead of providing the data type. One detail to note, in the schemas shown earlier, each field was marked either required or optional, but this makes no difference to how the field is encoded (nothing in the binary data indicates whether a field was required). The difference is simply that required enables a runtime check that fails if the field is not set, which can be useful for catching bugs.

A. Field Tags

Field tags are critical to the meaning of the encoded data. Each field is identified by its tag number (the numbers 1, 2, 3 in the schema) and annotated with a datatype. If a field value is not set, it is simply omitted from the encoded record. We can change the name of a field in the schema, since the encoded data never refers to field names, but we cannot change a field's tag, since that would make all existing encoded data invalid. We can add new fields to the schema, provided that we give each field a new tag number. If old code tries to read data written by new code, including a new field with a tag number it does not recognize, it can simply ignore that field. The datatype annotation allows the parser to determine how many bytes it needs to skip. This maintains forward compatibility: old code can read records that were written by new code. As long as each field has a unique tag number, new code can always read old data, because the tag numbers still have the same meaning. The only detail is that if we add a new field, we cannot make it required. If we were to add a field and make it required, that check would fail if new code read data written by old code, because the old code will not have written the new field that we added. Therefore, to maintain backward compatibility, every field we add after the initial deployment of the schema must be optional or have a default value. Removing a field is just like adding a field, with backward and forward compatibility concerns reversed. That means we can only remove a field that is optional, a required field can never be removed, and we can never use the same tag number again because we may still have data written somewhere that includes the old tag number, and that field must be ignored by new code.

B. Data types

With changing data types there is a risk that values will lose precision or get truncated. For example, say you change a 32-bit integer into a 64-bit integer, new code can easily read data written by old code, because the parser can fill in any missing bits with zeros. However, if old code reads data written by new code, the old code is still using a 32-bit variable to hold the value. If the decoded 64-bit value won't fit in 32 bits, it will be truncated. Protocol Buffers does not have a list or array datatype, but instead has a repeated marker for fields. This has the nice effect that it's okay to change an optional field into a repeated field. New code reading old data sees a list with zero or one elements, and old code reading new data sees only the last element of the list. Thrift has a dedicated list datatype, which is parameterized with the datatype of the list elements. This does not allow the same evolution from single-valued to multi-valued as Protocol Buffers does, but it has the advantage of supporting nested lists.

V. Avro

Apache Avro [4] is another binary encoding format that is interestingly different from Protocol Buffers and Thrift. It was started as a subproject of Hadoop, as a result of Thrift not being a good fit for Hadoop's use cases. Avro also uses a schema to specify the structure of the data being encoded. It has two schema languages: one (Avro IDL) intended for human editing, and one based on JSON that is more easily machine-readable. Example schema discussed above written in Avro IDL will look like below:

```
record Person {
    string              name;
    union {null, long} id = null;
    array<string>       sports;
}
```

and the equivalent JSON representation of that schema is:

```
{
    "type": "record",
    "name": "Person",
    "fields": [
        {
            "name": "name",
            "type": "string"
        },
        {
            "name": "id",
            "type": ["null", "long"],
            "default": null
        },
        {
            "name": "sports",
            "type": {
                "type": "array",
                "items": "string"
            }
        }
    ]
}
```

Avro doesn't tag numbers in the schema. If we encode our example object using this schema, the Avro binary encoding is just 17 bytes long which is the most compact of all the encodings we have seen. The encoding simply consists of values concatenated together. A string is just a length prefix followed by UTF-8 bytes, but there's nothing in the encoded data that tells that it is a string. It could just as well be an integer, or something else entirely. An integer is encoded using a variable-length encoding. To parse the binary data, we go through the fields in the order that they appear in the schema and use the schema to tell you the datatype of each field. This means that the binary data can only be decoded correctly if the code reading the data is using the exact same schema as the code that wrote the data. Any mismatch in the schema between the reader and the writer would mean incorrectly decoded data. Avro uses writer's and reader's schema for schema evolution. With Avro, when an application wants to encode some data, it encodes the data using whatever version of the schema it knows, that schema may be compiled into the application. This is known as the writer's schema. When an application wants to decode some data, it is expecting the data to be in some schema, which is known as the reader's schema. The key idea with Avro is that the writer's schema and the reader's schema don't have to be the same, they only need to be compatible. When data is decoded, Avro library resolves the differences by looking at the writer's schema and the reader's schema side by side and translating the data from the writer's schema into the reader's schema. The Avro specification [20] defines exactly how this resolution works. For example, it's no problem if the writer's schema and the reader's schema have their fields in a different order, because the schema resolution matches up the fields by field name. If the code reading the data encounters a field that appears in the writer's schema but not in the reader's schema, it is ignored. If the code reading the data expects some field, but the writer's schema does not contain a field of that name, it is filled in with a default value declared in the reader's schema.

A. Schema Evolution

With Avro, forward compatibility means that we can have a new version of the schema as writer and an old version of the schema as reader. Conversely, backward compatibility means that we can have a new version of the schema as reader and an old version as writer. To maintain compatibility, we may only add or remove a field that has a default value. For example, say we add a field with a default value, so this new field exists in the new schema but not the old one. When a reader using the new schema reads a record written with the old schema, the default value is filled in for the missing field. Changing the datatype of a field is possible, provided that Avro can convert the type. Changing the name of a field is possible: the reader's schema can contain aliases for field names, so it can match an old writer's schema field names against the aliases. This means that changing a field name is backward compatible but not forward compatible. Similarly, adding a branch to a union type is backward compatible but not forward compatible.

VI. Conclusion

Many services need to support rolling upgrades, where a new version of a service is gradually deployed to a few nodes at a time, rather than deploying to all nodes simultaneously. Rolling upgrades allow new versions of a service to be released without downtime and make deployments less risky allowing faulty releases to be detected and rolled back before they affect a large number of users. These properties are hugely beneficial for evolvability, the ease of making changes to an application. During rolling upgrades, or for various other reasons, we must assume that different nodes are running the different versions of our application's code. Thus, it is important that all data flowing around the system is encoded in a way that provides backward compatibility and forward compatibility. We discussed several data encoding formats and their compatibility properties starting from standard data encoding formats like JSON, XML, CSV and their compatibility and the problems faced through these textual data encoding formats. Binary schema–driven formats like Thrift, Protocol Buffers, and Avro allow compact, efficient encoding with clearly defined forward and backward compatibility semantics. The schemas can be useful for documentation and code generation in statically typed languages. However, they have the downside that data needs to be decoded before it is human-readable.

References

[1] P. Kulkarni, S. Deshpande, and S. Raj, "Object serialization in Java: A systematic approach," in Proceedings of the IEEE International Conference on Computer Science and Software Engineering, Beijing, China, 2008, pp. 235-240. DOI: 10.1109/ICCSSE.2008.13.
[2] P. Rodriguez and S. Krishnamurthy, "RESTful Web Services: A Framework for Scalable Web Applications," in Proceedings of the IEEE International Conference on Web Services (ICWS), San Francisco, CA, USA, 2008, pp. 608-615. DOI: 10.1109/ICWS.2008.
[3] B. S. Athey, R. E. Filman, and A. Goldberg, "An Introduction to the RPC Model," in Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS), Paris, France, 1982, pp. 137-143. DOI: 10.1109/ICDCS.1982.5345391.
[4] Apache Software Foundation "Apache Avro" - https://avro.apache.org/
[5] Apache Software Foundation "Apache Thrift" - https://thrift.apache.org/
[6] Protocol Buffers "Protobuf Dev" - https://protobuf.dev/
[7] MessagePack Specification - Binary JSON format - https://msgpack.org/
[8] Ruby Marshall Serialization Format - https://docs.ruby-lang.org/en/2.1.0/marshal_rdoc.html
[9] Python Object Serialization, https://docs.python.org/3/library/pickle.html, Pickle