Encoding and Evolution

Tuan Dong - 01/2024

Agenda

- Everything changes
- Compatibility
- Encoding - Binary encoding
- Tools
- Q&A

Everything changes

- Application change over time:
  - New products are launched
  - Features are added or modified (ex: upgrade app on Appstore)
  - User/business requirements changed (ex: do A/B testing)
- New platform replaces old platforms (ex: Microsoft Teams)

Everything changes

Systems need to evolve

So the underlying data schema also changes

Everything change

Suppose, we’re unicorns. Our application has need to:

- Support old-version customers without forcing them update
- Release a new feature without service downtime

How do you design a system that acquire the requirement?

Everything change

Old and new versions of the code,

Old and new data formats,

may potentially coexist in the system at the same time.

So need to ensure **compatibility** to not break these mess

Compatibility

Backward compatibility

> Newer code can read data that was written by older code.

Forward compatibility

> Older code can read data that was written by newer code.

Encoding

Program work with data in (at least) 2 different representations:

- In memory: kept in objects, structs, lists, arrays, hash tables, trees and so on.
- Self-contained sequence of bytes (write data to file, send data over network)

=> Need translation between the two

Encoding

As everyone known, 2 translation between the two format:

- From in-memory to a sequence of bytes: **encoding** (also known as **serialization** or **marshalling**)
- The reverse: decoding (parsing, deserialization, unmarshalling)

Encoding - Binary encoding

Textual JSON is the most popular because of human readable, widely adopted, simplicity

So, why we still have binary encoding?

- Efficient for data storage (want to minimize storage resource)
- Efficient for data transfer (want to reduce time for service communication)

For a small dataset, the gains are negligible. But once we get into the **terabytes**, **the choice of data format** can have **a big impact**.

Encoding - Binary encoding

Given a record

```
{
        "userName": "Martin",
        "favoriteNumber": 1337,
        "interests": ["daydreaming", "hacking"]
}
```

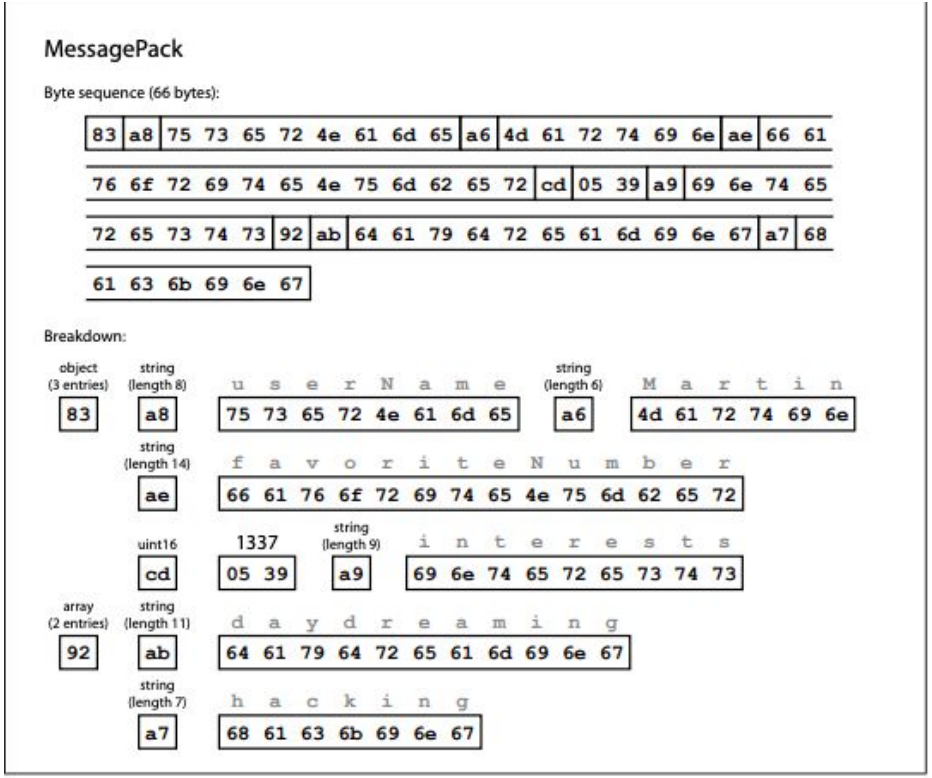
Textual JSON takes 81 bytes
We will discover some binary encodings

Encoding - Binary encoding

Some binary encodings for JSON

- MessagePack
- BSON (MongoDB)
- BJSON, UBJSON, BISON, …

MessagePack **reduces ~19%** from 81 bytes to 66 bytes



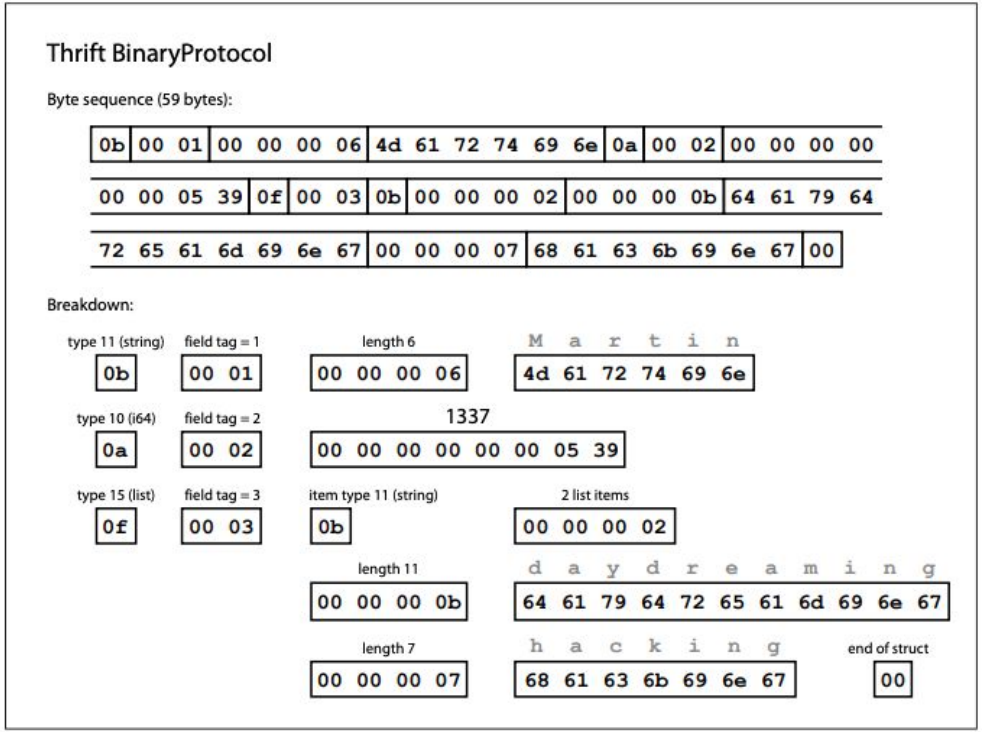
Encoding - Binary encoding

Thrift and Protocol Buffers

- Based on the same principle
- Thrift developed by Facebook
- Protocol Buffers developed by Google
- Both become open source in 2007-08
- Both require a **schema definition** for any data that is encoded (.proto)

Encoding - Binary encoding

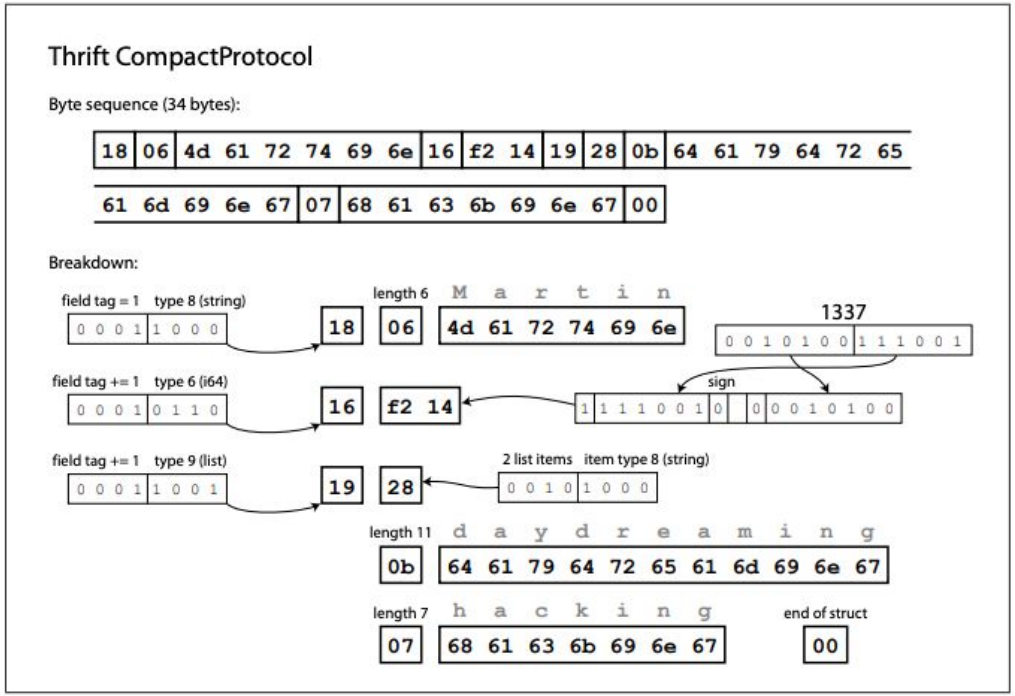
Thrift has 2 different binary encoding formats: **BinaryProtocol** and **CompactProtocol**



Encoding - Binary encoding

CompactProtocol use less bytes:

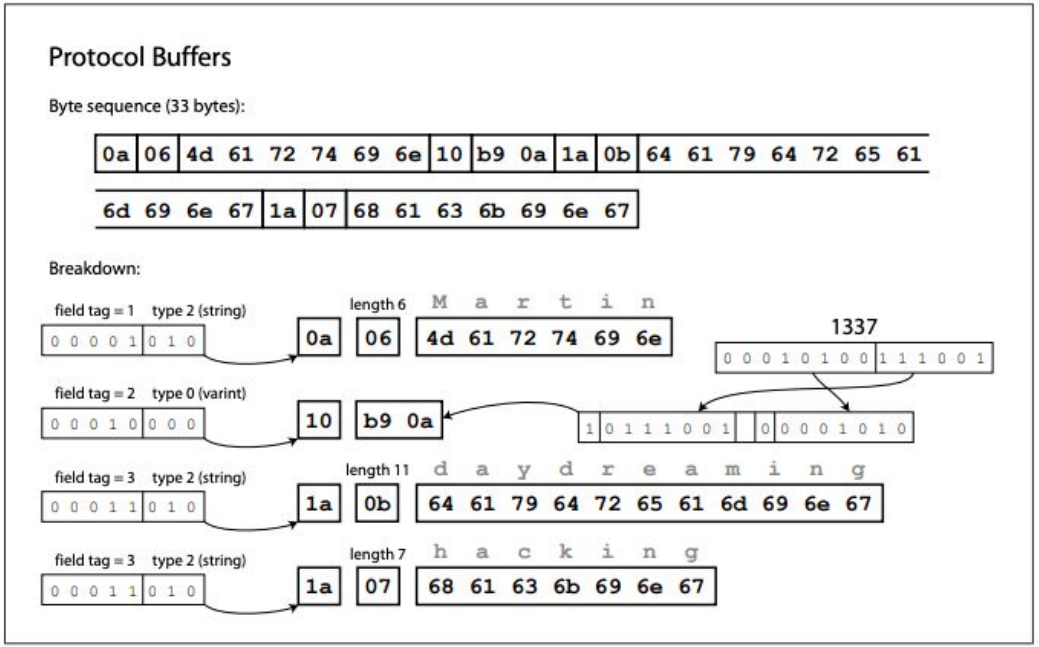
- Packing the field type and tag number into a single byte
- **Rather than using a full 8 bytes** for the number 1337, it is **encoded in two bytes**, using the top bit of each byte to indicate whether there are still more bytes to come.



Encoding - Binary Encoding

Protocol Buffers quite similar to Thrift's CompactProtocol

Both **reduces ~60%** from 81 bytes to 33 bytes



Encoding - Binary Encoding

How do Thrift and Protocol Buffers handle schema changes while keeping backward and forward compatibility?

Suppose we add a field with **a new tag number**:

- New code writes **encoded data** into database/file
- Old code read the encoded (which doesn’t know about the new tag number)
- The parser of old code can simply ignore that field

=> Maintains forward compatibility

Encoding - Binary Encoding

What about backward compatibility?

- If we don’t change any tag number, keep the old tag number still the same
- The new code can always read data written by old code

Remove a field?

- Just like add a field, still backward and forward compatibility
- Only remove optional field
- Never use the **tag number has been removed** again

Encoding - Binary Encoding

Change datatype of a field?

- Possible, but risk of values lose precision or get truncated.
- Ex: change a 32-bit integer into a 64-bit integer
- New code can easily read data written by old code, but old code only read 32 bits

Encoding - Binary Encoding

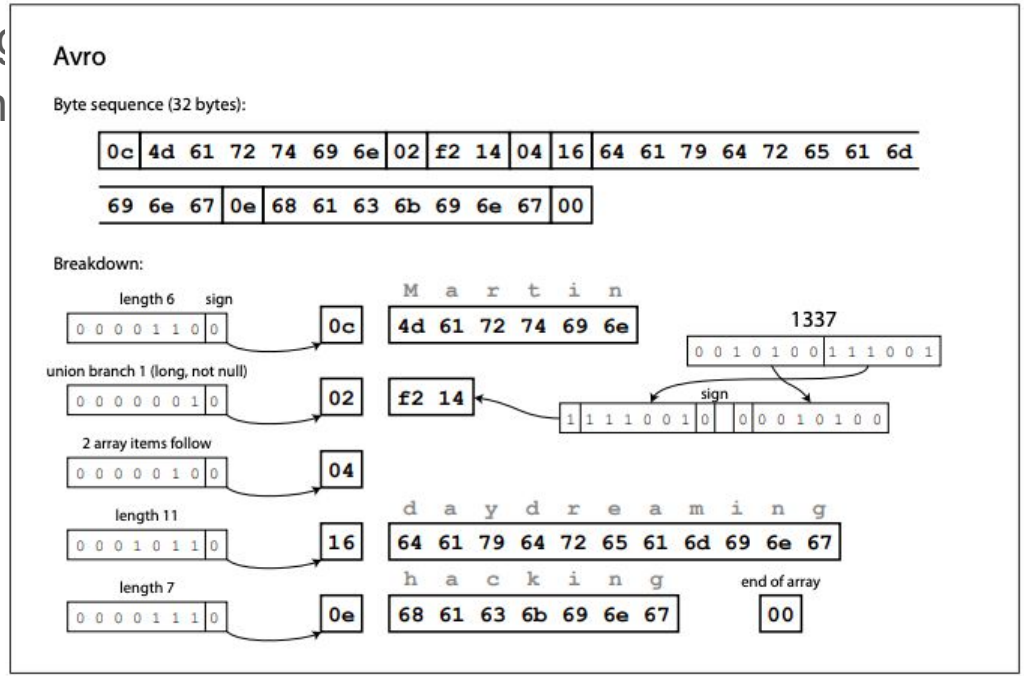
A curious fact of Protocol Buffers:

- **Not have a list or array data type**, but instead a **repeated** marker for fields
- The encoding of **a repeated field** is just: the same field tag **appears multiple times** in the record
- Nice effect: it’s ok to change an optional (single-valued) field into a repeated (multi-valued) field
- New code reading old data sees a list with **zero or one elements**
- Old code reading new data sees only the **last element** of the list

Encoding - Binary Encoding

Arvo is is another binary encoding format that is interestingly differen from Protocol Buffers and Thrift.

It was started in 2009 as a sub-project of Hadoop



Tools

Schema Registries in managing schema versions and ensuring data integrity

Kafka is the data storage has multiple-version records

Confluent Kafka has Schema Registry to manage these schemas

Q & A