Chapter 4 - Design Scalable systems

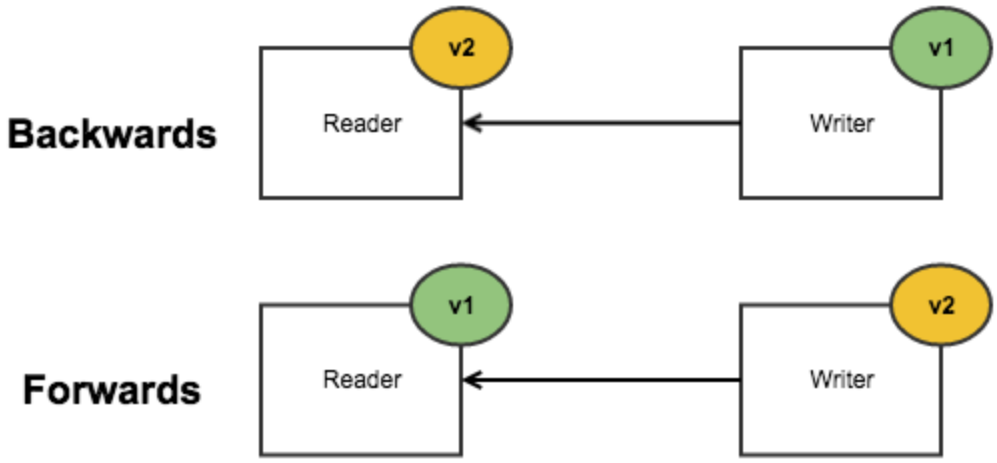
Type	Designing Data-Intensive Applicaitons
Content	
Created time	@October 21, 2022 5:20 PM
Last edited time	@October 22, 2022 3:56 PM
Resources	
💼 Projects	

Encoding and Evolution

💡 Applications change over time as features or business circumstances evolve. With this comes several challenges for a data engineer as changes to an application usually comes with changes to the data. Our systems must be able to handle forward and backward compatibility.

- This is a crucial concept in evolvability as mentioned in Chapter 1.

✏️ **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.



- Backward compatibility is usually easier to achieve as you author the new code you know the format written by the older code and can explicitly handle it.
- Forward compatibility is trickier as it requires older code to ignore additions made by a newer version of the code.

This chapter will look at several enconding formats, how they handle schema changes and how they support systems where new data and old data need to coexist. We will then discuss how these formats are used with data transfer protocols.

Formats for Encoding Data

Programs work with data in two different representations:

1. In memory, data is kept in objects etc… They are optimized for efficient access and manipulation by the CPU (typically using pointers)
2. When you want to write data to a file, or send it over the network it must be encoded into a **self-contained sequence of bytes** since a point wouldn't make sense to any other process, these sequences heavily differ from in memory representations.

- We therefore need some kind of translation between the two representations.

> 💡 The translation from in-memory representation to a byte-sequence is called **encoding (serialization),** and the reverse is called **decoding(deserialization).**

Language-specific formats

> 💡 Many programming languages come with built-in support for encoding in-memory objects into byte sequences. i.e. pickle for python

These encoding libraries are convenient due to the minimal overhead, however they have significant downsides:

- The encoding is tied to a particular program, and reading the data into another language is very difficult. If you store your data into a language specific format you are commiting yourself to that language for a long time.
- To restore data in the same object types, the decoding process needs to be able to instantiate arbitrary classes that can be a source of security problems. An attacker could get your app to decode an arbitrary byte sequence given them access to remotely execute code.
- Versioning data is not accounted for
- Bad performance in terms of efficiency

JSON, XML and Binary Variants

> 💡 These are standardized encodings that can be read and written by many programming languages. JSON and XML are popular but have issues to be considered.

- They are human readable which is a good trait

- They is a lot of ambiguity around the encoding of numbers, they cannot handle large numbers very well
- Good support for Unicode character strings but they don’t support binary strings
- Optional schema support
- CSV has no schema, so it is up to the app to define the meaning of each row and column
- Very popular as data interchange formats (sending from one organisation to another)

Binary encoding

> 💡 For data that is used internally within your organization, there is less pressure to use the most common format, and you can instead use the format that is faster and more compact. For small datasets the gains are negligible but for TB or PB the choice of data format can have large impacts.

- Both JSON and XML use a lot of space compared to binary formats
  - Since JSON/XML don’t prescibe schemas they need to include all the object field names within the encoded data. This adds space.

Thrift and Protocol Buffers

> 💡 Require a schema for any data that is encoded, they both come with a code generation tool that takes a schema definition and produces classes that implement the schema in various programming languages. Your applicaiton code can call this generated code to encode or decode records of the schema.

- The biggest difference in these formats are that there are no field names, instead they make use of field tags

- They also take up a lot less space meaning they are much more compact

> ✏️ **Field tags:** Field tags acts as aliases for fields, they are a compact way of saying which field we are referring to, without having to spell out the field name.

Field tags and schema evolution

> ✏️ **Schema evolution:** Schemas inevitably need to change over time

How does Thrift and Protocol Buffers handle schema changes without storing the field names?

- An encoded record is just the concatenation of it's encoded fields. Each field is identified by it's tag number and annotated with a datatype. If a field value is not set, it is omitted from the encoded record. It is clear that field tags are critical to the meaning of the encoded data.

> 💡 You can change the name of a field in the schema, since the encoded data never refers to field names, but you cannot change a field's tag, since that would make all existing encoded data invalid.

- Forward compatibility is easy to maintain as you can add new fields to the schema (provided you give each field a new tag number). If old code tries to read data with a field tag it doesn't recognize it simply ignores it.
- You can add new fields with a unique tag number, this ensures backward compatibility as new code always can read old data, the only detail is that new fields cannot be required, as checks would fail

Datatypes and schema evolution

💡 Data types can be changed however it comes with the risk of truncating data. 32-bit variable changed to a 64 bit-value will be truncated by old code.

Avro

💡 Avro is another binary encoding format, but is different to thrift and protocol buffers as it does not include tag numbers in the schema. It is the most compact of the binary encodings.

- To parse avro binary data you must go through the fields in the order that they appear in the schema. This means for avro to work correctly the data must be using the exact same schema as the code that wrote the data. Any mismatch will lead to errors.
- How does Avro support schema evolution?

The writer's schema and the reader's schema

✏️ **Writer's schema:** The schema used to encode the data, the schema compiled into the applicaiton for example.
Reader's schema: The schema used to decode the data, what the application expects the data to be in formatted in.

- The reader's schema is the schema the app code is relying on

💡 The key idea behind Avro is that the writer's and reader's schema don't have to be the same they just need to be **compatible**. When data is decoded the avro library resolves differences by looking at the writer's and reader's schema side by side, and translating the data from the writer's schema into the reader's schema.

- This concept allows avro to decode data if both schema's are in different orders, as avro matches up the fields by field name

    - Unmatched data will be ignored

Schema evolution rules

- Avro forward compatibility means that you can have a new version of the schema as writer and an old version as reader
- Conversly, backward compatibility is the opposite

Merits of schema

> 💡 For binary encoding formats, the schema's are much simpler than JSON or XML schemas. While textual data formats are usually simpler to use, binary encodings are a viable option with a nice number of properties:

- Can be much more compact
- Schema can be a valuable form of documentation
- Keeping a database of schemas allows you to check forward and backward compatibility of schema changes
- For users of statically typed programming languages the ability to generate code from the schema is useful, as it enables type checking at compile time

Modes of Dataflow

Common ways data flows through processes:

1. via Databases
2. via Service calls
3. via Asynchronous message passing

Dataflow through databases

> 💡 In a database the process that writes to the database encodes it, and the process that reads from it decodes it. There may be a single process accessing the database in which case the reader is simply a later version of the process. “Sending a message to your future self”.

- In a dynamic environment it is common that older and newer code will be accessing your database.
- Backward compatibility is necessary as new code should be able to decode data written by older code and same for forward compatability
- There are application level considerations to be made as well, as your code may have model objects that do not include new fields or changed data types that will not be written back into the database.

- [ ] watch a video on this

Dataflow through services: REST and RPC

> 💡 When processes need to communicate over a network one of the most common arrangements is to have two roles: clients and servers. Servers exposed an API over the network and clients connect to the API to make requests.