Designing Data-Intensive Applications

Chapter 4: Encoding and Evolution

Chapter Introduction

- There are many ways to encode data (JSON, XML, Protos, Thrift, Avro). Key questions that this chapter addresses:
    - How do they each handle schema changes and backward/forward compatibility?
    - How are each of these formats used for data storage and communication? (web services, REST, RPC, message passing - actors and queues)
- As we learn about these different types of encodings, keep in mind the following concept and consider how different encodings need to handle this problem:

Rolling out Changes

- Problem: Changes to a product's feature set will often change the data it stores (new fields, new types, etc.). With data format changes or schema changes, application code often needs an update. But the update often cannot happen instantly. How to update?
    1. Server-side Application Solution:
        - Perform a rolling upgrade: deploy the new version to a few nodes at a time. Continually check to see that the rollout is running smoothly.
        - Benefits: More frequent releases and evolvability.
    2. Client-side Application Solution: At the mercy of the user - they need to update!

⇒ This all results in multiple versions of the code and data formats coexisting at the same time! We need to maintain compatibility in both directions (backward and forward compatibility)

Notes

Formats for Encoding Data

Programs have two types of data:

In-memory data: data in objects, structs, lists, arrays, hash tables, trees, etc. Optimized for access and manipulation by the CPU.

Data to be sent over the network/stored on disk: This data is encoded in a self-contained sequence of bytes.

Key Question	Notes
How do we translate between in-memory data and data that needs to be sent over a network?	Translating the in-memory representation to a byte sequence is called encoding (aka. Serialization or marshalling). The reverse is called decoding.

What are the deep problems with programming languages that have their own encoding mechanism for in-memory data to byte sequences? Why is it not recommended to rely on these for anything other than transient purposes?	1. Encoding is tied to the programming language. Commitment to the language is required for a long time. Integration projects with other languages are limited. 2. To restore data in the same object type, the decoding process needs to instantiate arbitrary classes. Security issues may arise[1]. 3. These libraries may neglect backward/forward compatibility. Versioning is an afterthought. Meant for quick and easy encoding. 4. Might be inefficient. This could be an afterthought with these libraries.

JSON, XML, CSV

- Widely known and supported. Still - many challenges.
- Challenges:
    - XML and CSV number encodings - how to distinguish between a string and a number?
    - JSON: Number parsing for very large numbers. Case study: Twitter needs to encode two separate data points to capture the 64-bit tweet ID number: once as a number and once as a decimal string.
    - JSON and XML do not support binary data. The workaround is costly in terms of size.
    - XML/JSON: Complexity with schemas. If no schema, application needs to hardcode the encoding/decoding logic.
    - CSV: No schema. New rows or columns need to be manually handled. Not all implementations of CSV follow the same rules (eg. how to handle commas)

Binary Encoding

- The encoding format really matters when you work at a scale of terabytes.
- JSON: less verbose than XML but takes up a lot of space compared to binary formats
    - There are binary encoding options for XML and JSON
    - MessagePack can cut down the size, but there is a loss of human-readability. Any better options? Yes - Thrift and Protos.
- Apache Thrift
    - Two binary encoding formats: BinaryProtocol and Compact Protocol
    - Field tags: A compact way of saying which field we are talking about, without having to spell out the field name. This is an improvement over MessagePack.
    - The Compact Protocol version has some additional compaction strategies over the BinaryProtocol design.
- Protocol Buffers
    - Similar to Thrift's Compact Protocol.
    - Backward Compatibility: You cannot change a field tag - it would make all existing encoded data invalid. The tag is heavily used in the byte sequence (see Figure 4-4). Each new field needs a new tag number. New fields cannot be required, since new

[1] If an attacker can get your application to decode an arbitrary byte sequence, they can cause arbitrary classes to be called and this can allow them to remotely execute arbitrary code.

code cannot read data encoded by old code (no required data would have been encoded). To maintain backward compatibility, all fields after the initial deployment need to be optional or have a default value.
    - Forward Compatibility: Old code can ignore new tag numbers.
    - Protos allow an optional field to become repeated. New code with old data sees a list with zero or one elements. Old code reading new data sees only the last element of the list.
- Apache Avro: Binary encoding format different from Protos and Thrift.
    - Two schema languages: (1) one for human editing (2) one easily machine readable.
    - No tag numbers. Yet it is more compact!?
    - Encoding is just a list of values concatenated together.
    - To decode, you need the exact same schema. Schema is used to tell what data type to look at. Variable length data is supported.
    - Encoded data uses the “writer’s schema”: the schema used at write time. When it is read, it uses the “reader’s schema”: the schema the application code is relying on and it may have been generated during the build process.
        - Interestingly, these do not have to be the same schema, they only need to be compatible. Differences are resolved. See figure 4-6.
    - To maintain compatibility, fields can only be added or removed if there is a default value. Example: If a reader has a new schema and reads a record encoded with the old schema, the default value will be filled in.
    - There are backward compatibility limitations when it comes to changing field names and “union types”.
    - How does the reader know what the writer’s schema was? In order to resolve differences, the reader needs that writer schema.
        - Writers schema could exist at the top of the file. A common use of Avro is in Hadoop with millions of records of the same schema.
        - Or - a version number for the writers schema may be included with the record. The reader can fetch the version number then lookup and fetch the right writers schema.
        - Or - the schema version can be communicated at the beginning of a connection setup. See Avro RPC Protocol.
    - A database of schemas would be useful to have in whichever option is used - documentation and a chance to check schema compatibility(?)
    - Avro is great for dynamically generating schemas. For example, if you have a relational database whos content needs to be dumped in a file.
    - Avro provides optional code generation for statically typed languages (Java, C++, C#) but it can also be used without any code generation. In contrast, Thrift and Protos rely on code generation. Avro is self-describing the way that JSON is - just open the file with an Avro library and read the data.
- Summary: Binary encodings are more compact, schemas provide documentation (and it is always up to date because that is required to decode), keeping a database of schemas allows for checking backward and forward compatibility, code generation is useful for type checking at compile time.

Models of Dataflow: How does data flow between processes?

- Dataflow through Databases
    - Backward compatibility is necessary. There might be only one process writing and reading from the database. Future versions of that process and database need to read old data too.
    - Forward compatibility is also necessary. Many processes will read and write to the database. Some processes may have gotten the latest update and are writing new data to the database with newer code. Then, older code could read that data.
    - Snag: What if new code writes a new field. Then old code updates that record. How should it handle this new field? It will be unknown.
    - “Data outlives code” - schema evolution must help code maintain compatibility with old data
- Dataflow through Services: REST and RPC
    - A key design goal of a service-oriented architecture is to make the application easier to change and maintain by making services independently deployable and evolable.
        - This means that there could be old and new versions of servers running at the same time, often across many teams. Data encapsulated across clients and servers must be compatible across these versions.
    - SOAP (alternative to REST, XML-based) has fallen out of favor in smaller companies due to interoperability challenges.
    - The RPC model tries to make a request to a remote network service look the same as calling a function or method in your programming language (location transparency) - but there are differences:
        - Network requests are unpredictable - problems outside your control happen and you need to anticipate them
        - Timeouts can happen (but can’t happen with a local call) and are challenging to debug.
        - Requests might be processed but the response isn’t getting through. Need idempotence otherwise there will be duplications of efforts.
        - Network calls have higher latency
        - Larger objects can’t be efficiently passed with a network call. But they can be with a pointer in a local call.
        - Challenges with datatype compatibility across languages can crop up if the client and server are in different languages.
    - Streams: a call consists of not just one request and one response, but a series of requests and responses over time
    - Promises: encapsulates async actions that may fail. Simplifies parallelization.
    - RPC protocols with a binary encoding have better performance than generic JSON over REST. But RESTful APIs have other advantages: great for experimentation and debugging, widely supported, lots of tools.
    - RPC frameworks are mainly used on requests between services in the same organization. REST is predominant for public APIs.

    - Backward and forward compatibility properties of an RPC are inherited from the encoding it uses.
        - Note: RPC is often used for communication across the organization. Since the service has no control over clients and cannot force an upgrade, multiple versions of the service API might need to be supported.
        - Versioning can be used for REST. Version tracking can happen in a database for users with an API key or tracked in the request header.
- Dataflow through Message-Passing
    - Messages are sent to a “Queue” or “Topic”. Message broker ensures that the message is delivered to 1+ “consumers” or “subscribers” to that topic.
    - Async message passing systems are somewhere in between RPCs and databases. Benefits:
        - Recipient might be overloaded and this acts as a buffer, improves reliability
        - Prevents messages from being lost: Can redeliver messages to a process that has crashed
        - The sender doesn’t need to know the IP of the receiver. Useful if the IP keeps changing.
        - One message can be sent to many recipients.
        - Decouples sender and receiver.
    - Data is sent via a message broker - an intermediary that is NOT a network connection, but a temporary store of the message.
    - The sender does not wait for the message to be delivered.
    - Consumers of a topic might even process the message and enqueue to another topic! Or it could enqueue to a response queue so that the original sender gets a reply.
    - You can use any encoding format. As long as that encoding is backward and forward compatible, publishers and consumers can be deployed independently and rolled out.
    - Distributed actor frameworks: logic is encapsulated in actors, not threads. The application is scaled across many nodes. Location transparency is supported: the sender and receiver can be on the same node or different nodes, and it works better than in RPC because the actor model assumes that messages may be lost. <<< more examples needed here. I don’t fully understand why the actor models handling of messages being lost is better for location transparency.