**Chapter 4: Encoding and Evolution**

**Shema on read**: “schema-on-read (“schemaless”) databases don’t enforce a schema, so the database can contain a mixture of older and newer data formats written at different times”

When an application changes, the database often changes (e.g a new record-type or field is added and needs to be handled)

* **Backward compatibility**: Newer code can read data that was written by older code
  + Easy to achieve by simply keeping the old code
* **Forward compatibility**: Older code can read data that was written by newer code
  + Harder to achieve with older code needing to ignore the additions made by newer code

**Chapter overview**

“In this chapter we will look at several formats for encoding data, including JSON, XML, Protocol Buffers, Thrift, and Avro. In particular, we will look at 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), and remote procedure calls (RPC), as well as message-passing systems such as actors and message queues.”

**Formats for encoding data**

In memory data is stored as objects, structs, lists, hashmaps, etc

Data stored on disk or transfered over the network is encoded as a self-contained sequence of bytes

“The translation from the in-memory representation to a byte sequence is called encoding (also known as serialization or marshalling), and the reverse is called decoding (parsing, deserialization, unmarshalling).”

**Language specific formats**

“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 [2], Python has pickle [3], and so on.”

**JSON, XML, and Binary Variants**

JSON, XML, and CSV are common text-based encoding types

Some issues:

* Encoding of numbers (distinguishing between ints, strings, floats)
* Handling large numbers (some applications are unable to handle numbers larger than 2\*\*53, and that is not accounted for in the encoding schemes
* Binary strings without a character encoding aren’t supported. Base64 encoding often used as a workaround, but application schema must indicate the value is b64 encoded. Also increases data size by 33%
* CSVs don’t have any schema

**Binary encoding**

There are lots of binary encodings for json (MessagePack, BSON, BJSON, UBJSON, BISON) and xml (WBXML and Fast Infoset)

Example of binary encoding of json using MessagePack

A text on a white background

Description automatically generated

A screenshot of a computer

Description automatically generated

**Thrift and Protocol Buffers**

“Apache Thrift [15] and Protocol Buffers (protobuf) [16] 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 [17].”

Thrift interface definition language (IDL) example:

A computer code with blue text

Description automatically generated

A white background with black text and blue and orange numbers

Description automatically generated

“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 [18]. Your application code can call this generated code to encode or decode records of the schema.”

A screenshot of a computer

Description automatically generated

**Thrift BinaryProtocol** encoding scheme contains a binary representation of a type annotation, and instead of field names it contains the field tag (the number defined in the schema definition)

**Thrift CompactProtocol** adds additional compression by using variable length integers and packing field tag and type into a single byte

A screenshot of a computer program

Description automatically generated

**Protocol Buffers** is very similar to Thrift CompactProtocol

A screenshot of a computer program

Description automatically generated

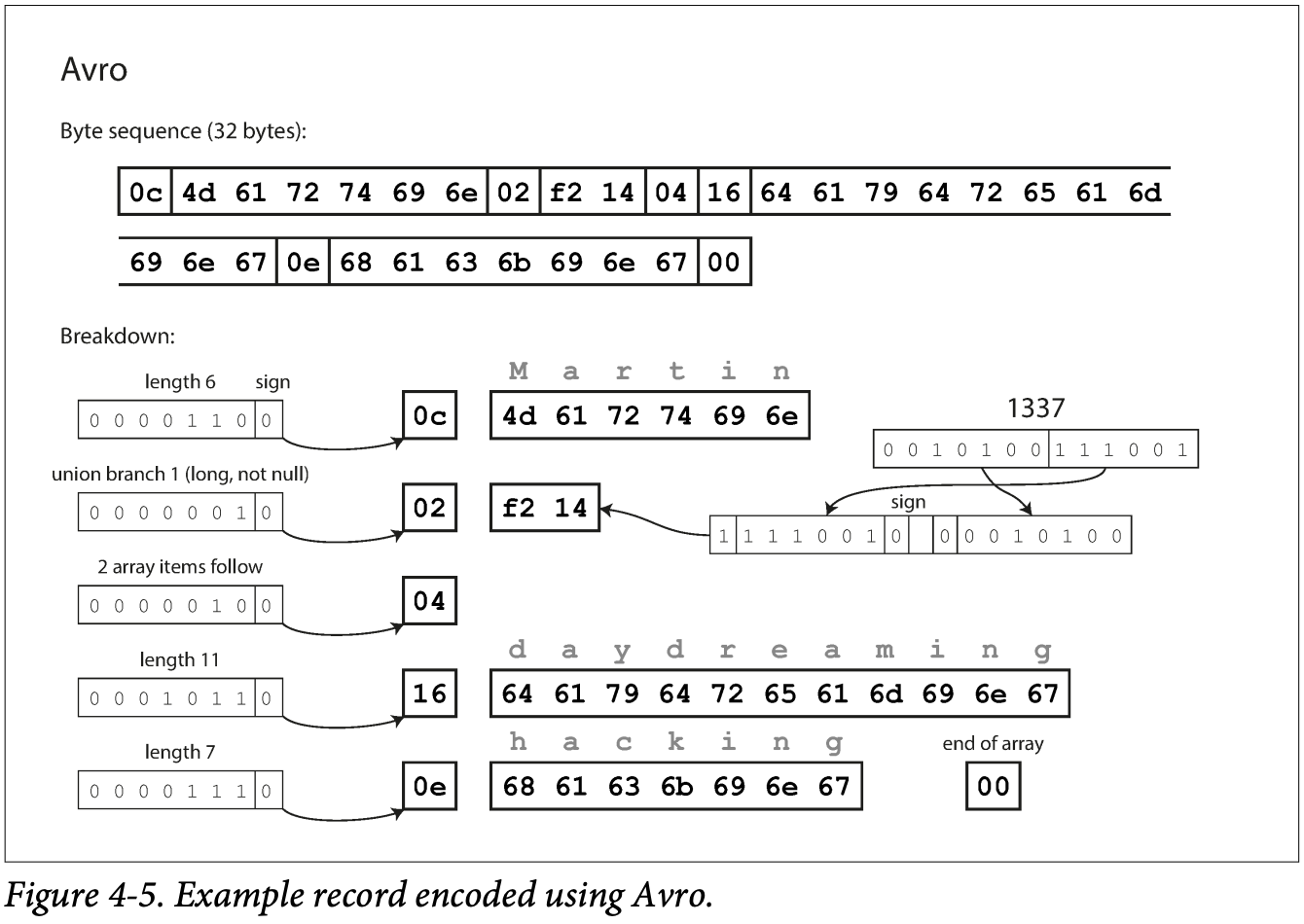
**Note**: Making a field required is not encoded into the binary encoding, but enables a runtime check that fails if the field is missing

**Avro**

“Apache Avro [20] is another binary encoding format that is interestingly different from Protocol Buffers and Thrift. It was started in 2009 as a subproject of Hadoop, as a result of Thrift not being a good fit for Hadoop’s use cases [21].”

A computer code with text

Description automatically generated with medium confidence



**Writer’s schema and reader’s schema**

**Writer’s schema**: “when an application wants to encode some data (to write it to a file or database, to send it over the network, etc.), it encodes the data using whatever version of the schema it knows about”

**Reader’s schema**: “When an application wants to decode some data (read it from a file or database, receive it from the network, etc.), it is expecting the data to be in some schema… That is the schema the application code is relying on —code may have been generated from that schema during the application’s build process”

These do not have to match exactly, but must be compatible

* Order of fields doesn’t have to be same
* If the application reading the data doesn’t recognize a field in the writer’s schema, it is ignored
* If the reader expects a field not in the writer’s schema, it is filled with a default

**Schema evolution with Avro**

“With Avro, forward compatibility means that you can have a new version of the schema as writer and an old version of the schema as reader. Conversely, backward compatibility means that you can have a new version of the schema as reader and an old version as writer.”

Default values are important to not break forward and backward compatibility (by adding or removing a field with no default value)

Avro uses the *union* type to allow nulls, and uses union types and default values instead of *required* and *optional* field types

union { null, long, string } field;

**Schema versioning**

A record is kept of each schema version when schema changes take place, and the schema version can be kept as an attribute of each record. When transferring data over a bi-directional network connection, the sender and receiver negotiate a schema version to use for the session.

“A database of schema versions is a useful thing to have in any case, since it acts as documentation and gives you a chance to check schema compatibility”

**Dynamically generated schemas**

A big advantage of Avro over Thrift and Protobuf is that it can generate schemas dynamically, since it doesn’t use field tags (so field names are simply mapped to column names when exporting data from a db). If field tags were used, there would need to be metadata tables involved and potentially manually managed.

**Modes of Dataflow**

We’ve discussed that when you want to send data to a process with which you don’t share memory, you need to encode the data in some way (JSON, XML, Protobuf, Thrift, Avro, etc)

We talked about schema evolution and forward and backward compatibility

Now we’ll discuss some common forms of data exchange between processes:

* Databases
* Service calls (REST and RPC)
* Asynchronous message passing

**Dataflow through databases**

“In a database, the process that writes to the database encodes the data, and the process that reads from the database decodes it. There may just be a single process accessing the database, in which case the reader is simply a later version of the same process—in that case you can think of storing something in the database as sending a message to your future self. Backward compatibility is clearly necessary here; otherwise your future self won’t be able to decode what you previously wrote.”

**Different values written at different times**

*Data outlives code:* Updating the code in an application is often done in a matter of minutes, however the data written by that application in the past is not usually updated along with the code

*Migrations* to a new schema can be done but are costly, and most relational databases allow schema changes such as adding new columns with a null value as the default.

**Dataflow Through Services: REST and RPC**

*Clients and servers*: “The servers expose an API over the network, and the clients can connect to the servers to make requests to that API. The API exposed by the server is known as a service.”

A server can also be a client to another service (like a database)

*service-oriented architecture* (SOA) or *microservices* *architecture*: larger applications are composed of smaller services with narrow areas of functionality

as opposed to a database, which allows for arbitrary queries, “services expose an application-specific API that only allows inputs and outputs that are predetermined by the business logic (application code) of the service”

A key design goal of a service-oriented/microservices architecture is to make the application easier to change and maintain by making services independently deployable and evolvable. For example, each service should be owned by one team, and that team should be able to release new versions of the service frequently, without having to coordinate with other teams. In other words, we should expect old and new versions of servers and clients to be running at the same time, and so the data encoding used by servers and clients must be compatible across versions of the service API— precisely what we’ve been talking about in this chapter.”

**Web services**

“When HTTP is used as the underlying protocol for talking to the service, it is called a web service”

Some examples

1. A client application running on a user’s device (e.g., a native app on a mobile device, or JavaScript web app using Ajax) making requests to a service over HTTP. These requests typically go over the public internet.
2. One service making requests to another service owned by the same organization, often located within the same datacenter, as part of a service-oriented/microser‐ vices architecture. (Software that supports this kind of use case is sometimes called middleware.)
3. One service making requests to a service owned by a different organization, usu‐ ally via the internet. This is used for data exchange between different organiza‐ tions’ backend systems. This category includes public APIs provided by online services, such as credit card processing systems, or OAuth for shared access to user data.

Two main approaches to web services: *REST* and *SOAP*

“REST is not a protocol, but rather a design philosophy that builds upon the principles of HTTP [34, 35]. It emphasizes simple data formats, using URLs for identifying resources and using HTTP features for cache control, authentication, and content type negotiation. REST has been gaining popularity compared to SOAP, at least in the context of cross-organizational service integration [36], and is often associated with microservices [31].

By contrast, SOAP is an XML-based protocol for making network API requests. Although it is most commonly used over HTTP, it aims to be independent from HTTP and avoids using most HTTP features. Instead, it comes with a sprawling and complex multitude of related standards (the web service framework, known as WS-\*) that add various features”

**The problems with remote procedure calls (RPCs)**

“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”

This seems convenient but has a lot of issues because of the major differences between local function calls and network requests

RPC frameworks have been built on top of all the encodings mentioned in this chapter (Avro, Thrift, Protobuf)

REST is the predominant style for public APIs

RPC frameworks are focused mainly on requests between services owned by the same organization, typically within the same datacenter

**Schema evolution in services**: the rules for schema changes are set by the encoding used by the service (e.g. JSON, XML, Avro, Protobuf, Thrift rules described above for forward and backward compatibility)

API versioning can be specified by the client via a version number specified in the url, an HTTP accept header, or an administrative interface for APIs that use an API key for client identification

**Message-Passing Dataflow**

***Asyncronous message-passing* systems** “are somewhere between RPC and databases. They are similar to RPC in that a client’s request (usually called a *message*) is delivered to another process with low latency. They are similar to databases in that the message is not sent via a direct network connection, but goes via an intermediary called a *message broker* (also called a *message queue* or *message-oriented middleware*), which stores the message temporarily.

Using a message broker has several advantages compared to direct RPC:

* It can act as a buffer if the recipient is unavailable or overloaded, and thus improve system reliability.
* It can automatically redeliver messages to a process that has crashed, and thus prevent messages from being lost.
* It avoids the sender needing to know the IP address and port number of the recipient (which is particularly useful in a cloud deployment where virtual machines often come and go).
* It allows one message to be sent to several recipients.
* It logically decouples the sender from the recipient (the sender just publishes messages and doesn’t care who consumes them).”

The communication pattern here is *asynchronous*, since it doesn’t wait for a message to be delivered before sending more messages

Messages are sent to a named *queue* or *topic*, and the message broker handles delivery of messages to *consumers* or *subscribers* of that topic

A topic provides one-way data flow, but consumers can pass received messages to another topic, so message queues can be chained together

Messages can use any encoding format, since they are treated as a sequence of bytes, and message brokers don’t enforce a schema of any kind

**Distributed actor frameworks**

“The *actor model* is a programming model for concurrency in a single process.”

* An actor represents one client, with logic encapsulated in the actor rather than dealing directly with threads
* May have local state not shared with other actors
* Communicates with other actors by sending or receiving asynchronous messages
* Actors only process one message at a time, so no need for dealing with threads

In *distributed actor frameworks*, this programming model is scaled across multiple nodes. If actors are on separate nodes, messages are encoded into byte sequences and shared across the network

3 common distributed actor frameworks

Akka

Orleans

Erlang OTP

**Summary**

“In this chapter we looked at several ways of turning data structures into bytes on the network or bytes on disk. We saw how the details of these encodings affect not only their efficiency, but more importantly also the architecture of applications and your options for deploying them.”

We covered rolling upgrades, where a new version of an application is deployed gradually to several nodes at a time, rather than to all nodes at once. This kind of deployment requires forward and backward compatibility, as new code may be reading data written by old code and vice-versa.

We discussed several common encoding formats:

* Programming language specific (e.g. pickle)
* Text based (e.g. JSON, XML)
* Binary formats (e.g. Thrift, Avro, Protobuf)

We discussed several common modes of dataflow:

* Databases
* REST and RPC APIs
* Async message passing