Chapter 5. Encoding and Evolution

Everything changes and nothing stands still.

—Heraclitus of Ephesus, as quoted by Plato in *Cratylus* (360 BCE)

Applications inevitably change over time. Features are added or modified as new products are launched, user requirements become better understood, or business circumstances change. In Chapter 2 we introduced the idea of *evolvability*: we should aim to build systems that make it easy to adapt to change (see "Evolvability: Making Change Easy").

In most cases, a change to an application's features also requires a change to data that it stores: perhaps a new field or record type needs to be captured, or perhaps existing data needs to be presented in a new way.

The data models we discussed in Chapter 3 have different ways of coping with such change. Relational databases generally assume that all data in the database conforms to one schema: although that schema can be changed (through schema migrations; i.e., `ALTER` statements), there is exactly one schema in force at any one point in time. By contrast, 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 (see "Schema flexibility in the document model").

When a data format or schema changes, a corresponding change to application code often needs to happen (for example, you add a new field to a record, and the application code starts reading and writing that field). However, in a large application, code changes often cannot happen instantaneously:

- With server-side applications you may want to perform a *rolling upgrade* (also known as a *staged rollout*), deploying the new version to a few nodes at a time, checking whether the new version is running smoothly, and gradually working your way through all the nodes. This allows new versions to be deployed without service downtime, and thus encourages more frequent releases and better evolvability.
- With client-side applications you're at the mercy of the user, who may not install the update for some time.

This means that old and new versions of the code, and old and new data formats, may potentially all coexist in the system at the same time. In order for the system to continue running smoothly, we need to maintain compatibility in both directions:

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 normally not hard to achieve: as author of the newer code, you know the format of data written by older code, and so you 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.

Another challenge with forward compatibility is illustrated in Figure 5-1. Say you add a field to a record schema, and the newer code creates a record containing that new field and stores it in a database. Subsequently, an older version of the code (which doesn't yet know about the new field) reads the record, updates it, and writes it back. In this situation, the desirable behavior is usually for the old code to keep the new field intact, even though it couldn't be interpreted. But if the record is decoded into a model object that does not explicitly preserve unknown fields, data can be lost, like in Figure 5-1.

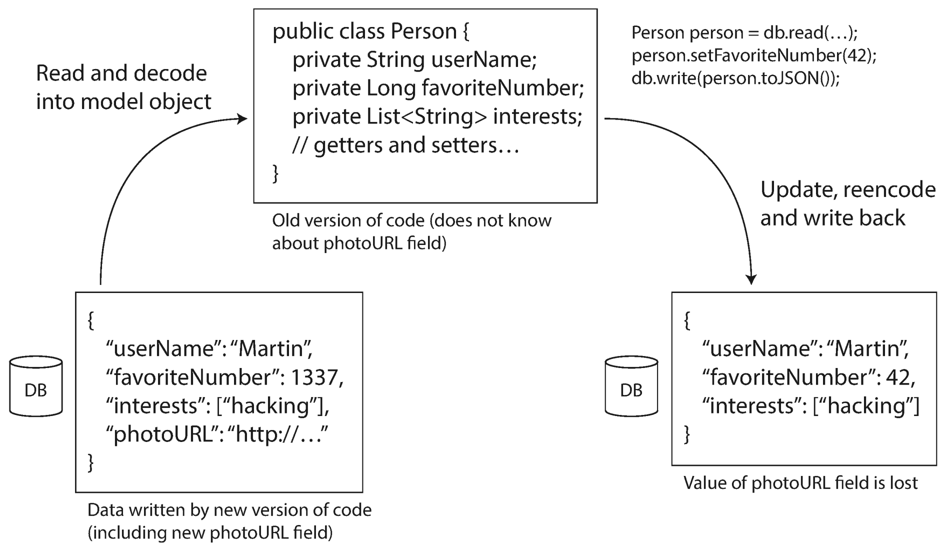

Read and decode into model object
public class Person {
private String userName;
private Long favoriteNumber;
private List<String> interests;
// getters and setters…
}
Old version of code (does not know about photoURL field)
Person person = db.read(…);
person.setFavoriteNumber(42);
db.write(person.toJSON());
Update, reencode and write back
DB
{ "userName": "Martin", "favoriteNumber": 1337, "interests": ["hacking"], "photoURL": "http://…" }
Data written by new version of code (including new photoURL field)
DB
{ "userName": "Martin", "favoriteNumber": 42, "interests": ["hacking"] }
Value of photoURL field is lost

Figure 5-1. When an older version of the application updates data previously written by a newer version of the application, data may be lost if you're not careful.

In this chapter we will look at several formats for encoding data, including JSON, XML, Protocol Buffers, 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 databases, web services, REST APIs, remote procedure calls (RPC), workflow engines, and event-driven systems such as actors and message queues.

Formats for Encoding Data

Programs usually work with data in (at least) two different representations:

1. In memory, data is kept in objects, structs, lists, arrays, hash tables, trees, and so on. These data structures 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, you have to encode it as some kind of self-contained sequence of bytes (for example, a JSON document). Since a pointer wouldn't make sense to any other process, this sequence-of-bytes representation often looks quite different from the data structures that are normally used in memory.

Thus, we need some kind of translation between the two representations. 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*).

TERMINOLOGY CLASH

Serialization is unfortunately also used in the context of transactions (see Chapter 8), with a completely different meaning. To avoid overloading the word we'll stick with *encoding* in this book, even though *serialization* is perhaps a more common term.

There are exceptions in which encoding/decoding is not needed—for example, when a database operates directly on compressed data loaded from disk, as discussed in "Query Execution: Compilation and Vectorization". There are also *zero-copy* data formats that are designed to be used both at runtime and on disk/on the network, without an explicit conversion step, such as Cap'n Proto and FlatBuffers.

However, most systems need to convert between in-memory objects and flat byte sequences. As this is such a common problem, there are a myriad different libraries and encoding formats to choose from. Let's do a brief overview.

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`, Python has `pickle`, Ruby has `Marshal`, and so on. Many third-party libraries also exist, such as Kryo for Java.

These encoding libraries are very convenient, because they allow in-memory objects to be saved and restored with minimal additional code. However, they also have a number of deep problems:

- The encoding is often tied to a particular programming language, and reading the data in another language is very difficult. If you store or transmit data in such an encoding, you are committing yourself to your current programming language for potentially a very long time, and precluding integrating your systems with those of other organizations (which may use different languages).

- In order to restore data in the same object types, the decoding process needs to be able to instantiate arbitrary classes. This is frequently a source of security problems [1]: if an attacker can get your application to decode an arbitrary byte sequence, they can instantiate arbitrary classes, which in turn often allows them to do terrible things such as remotely executing arbitrary code [2, 3].
- 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 [4].
- Efficiency (CPU time taken to encode or decode, and the size of the encoded structure) is also often an afterthought. For example, Java's built-in serialization is notorious for its bad performance and bloated encoding [5].

For these reasons it's generally a bad idea to use your language's built-in encoding for anything other than very transient purposes.

JSON, XML, and Binary Variants

When moving to standardized encodings that can be written and read by many programming languages, JSON and XML are the obvious contenders. They are widely known, widely supported, and almost as widely disliked. XML is often criticized for being too verbose and unnecessarily complicated [6]. JSON's popularity is mainly due to its built-in support in web browsers and simplicity relative to XML. CSV is another popular language-independent format, but it only supports tabular data without nesting.

JSON, XML, and CSV are textual formats, and thus somewhat human-readable (although the syntax is a popular topic of debate). Besides the superficial syntactic issues, they also have some subtle problems:

- 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 (except by referring to an external schema). JSON distinguishes strings and numbers, but it doesn't distinguish integers and floating-point numbers, and it doesn't specify a precision.
  This is a problem when dealing with large numbers; for example, integers greater than 2^{53} 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, such as JavaScript

[7]. An example of numbers larger than 2^{53} occurs on X (formerly Twitter), which uses a 64-bit number to identify each post. The JSON returned by the API includes post 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 [8].

- JSON and XML have good support for Unicode character strings (i.e., human-readable text), but they don't support binary strings (sequences of bytes without a character encoding). 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. This works, but it's somewhat hacky and increases the data size by 33%.
- XML Schema and JSON Schema are powerful, and thus quite complicated to learn and implement. Since the correct interpretation of data (such as numbers and binary strings) depends on information in the schema, applications that don't use XML/JSON schemas need to potentially hard-code the appropriate encoding/decoding logic instead.
- 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, you have to handle that change manually. CSV is also a quite vague format (what happens if a value contains a comma or a newline character?). Although its escaping rules have been formally specified [9], 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 (i.e., for sending data from one organization to another). 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. The difficulty of getting different organizations to agree on *anything* outweighs most other concerns.

JSON Schema

JSON Schema has become widely adopted as a way to model data whenever it's exchanged between systems or written to storage. You'll find JSON schemas in web services (see "Web services") as part of the OpenAPI web service specification, schema registries such as Confluent's Schema Registry and Red Hat's Apicurio Registry, and in databases such as PostgreSQL's pg_jsonschema validator extension and MongoDB's `$jsonSchema` validator syntax.

The JSON Schema specification offers a number of features. Schemas include standard primitive types including strings, numbers, integers, objects, arrays, booleans, or nulls. But JSON Schema also offers a separate validation specification that allows developers to overlay constraints on fields. For example, a `port` field might have a minimum of 1 and a maximum of 65535.

JSON Schemas can have either open or closed content models. An open content model permits any field not defined in the schema to exist with any data type, whereas a closed content model only allows fields that are explicitly defined. The open content model in JSON Schema is enabled when `additionalProperties` is set to `true`, which is the default. Thus, JSON Schemas are usually a definition of what *isn't* permitted (namely, invalid values on any of the defined fields), rather than what *is* permitted in a schema.

Open content models are powerful, but can be complex. For example, say you want to define a map from integers (such as IDs) to strings. JSON does not have a map or dictionary type, only an "object" type that can contain string keys, and values of any type. You can then constrain this type with JSON Schema so that keys may only contain digits, and values can only be strings, using `patternProperties` and `additionalProperties` as shown in Example 5-1.

Example 5-1. Example JSON Schema with integer keys and string values. Integer keys are represented as strings containing only integers since JSON Schema requires all keys to be strings.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "patternProperties": {
    "^[0-9]+$": {
      "type": "string"
    }
  },
  "additionalProperties": false
}
```

In addition to open and closed content models and validators, JSON Schema supports conditional if/else schema logic, named types, references to remote schemas, and much more. All of this makes for a very powerful schema language. Such features also make for unwieldy definitions. It can be challenging to resolve remote schemas, reason about conditional rules, or evolve schemas in a forwards or backwards compatible way [10]. Similar concerns apply to XML Schema [11].

Binary encoding

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 profusion of binary encodings for JSON (MessagePack, CBOR, BSON, BJSON, UBJSON, BISON, Hessian, and Smile, to name a few) and for XML (WBXML and Fast Infoset, for example). These formats have been adopted in various niches, as they are more compact and sometimes faster to parse, but none of them are as widely adopted as the textual versions of JSON and XML [12].

Some of these formats extend the set of datatypes (e.g., distinguishing integers and floating-point numbers, or adding support for binary strings), but otherwise they keep the JSON/XML data model unchanged. In particular, since they don't prescribe a schema, they need to include all the object field names within the encoded data. That is, in a binary encoding of the JSON document in Example 5-2, they will need to include the strings `userName`, `favoriteNumber`, and `interests` somewhere.

Example 5-2. Example record which we will encode in several binary formats in this chapter

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

Let's look at an example of MessagePack, a binary encoding for JSON. Figure 5-2 shows the byte sequence that you get if you encode the JSON document in Example 5-2 with MessagePack. The first few bytes are as follows:

1. The first byte, `0x83`, indicates that what follows is an object (most significant four bits = `0x80`) with three fields (least significant four bits =

`0x03`). (In case you're wondering what happens if an object has more than 15 fields, so that the number of fields doesn't fit in four bits, it then gets a different type indicator, and the number of fields is encoded in two or four bytes.)

2. The second byte, `0xa8` , indicates that what follows is a string (most significant four bits = `0xa0`) that is eight bytes long (least significant four bits = `0x08`).
3. The next eight bytes are the field name `userName` in ASCII. Since the length was indicated previously, there's no need for any marker to tell us where the string ends (or any escaping).
4. The next seven bytes encode the six-letter string value `Martin` with a prefix `0xa6` , and so on.

The binary encoding is 66 bytes long, which is only a little less than the 81 bytes taken by the textual JSON encoding (with whitespace removed). All the binary encodings of JSON are similar in this regard. It's not clear whether such a small space reduction (and perhaps a speedup in parsing) is worth the loss of human-readability.

In the following sections we will see how we can do much better, and encode the same record in just 32 bytes.

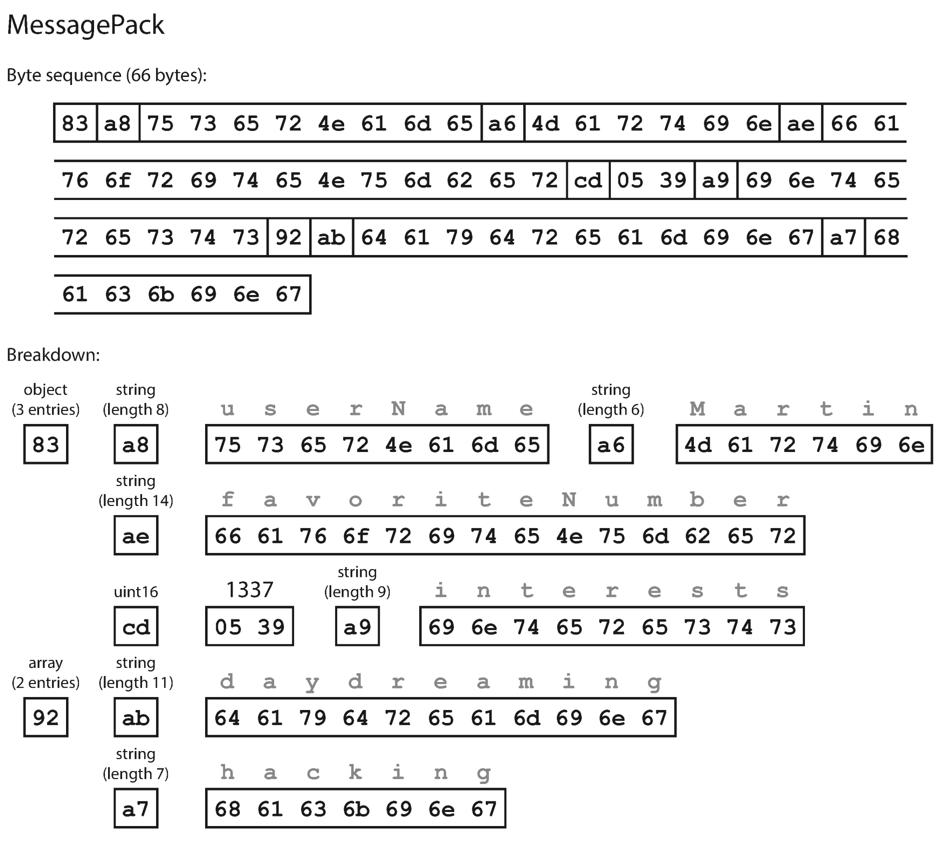


Figure 5-2. Example record (Example 5-2) encoded using MessagePack.

Protocol Buffers

Protocol Buffers (protobuf) is a binary encoding library developed at Google. It is similar to Apache Thrift, which was originally developed by Facebook [13]; most of what this section says about Protocol Buffers applies also to Thrift.

Protocol Buffers requires a schema for any data that is encoded. To encode the data in Example 5-2 in Protocol Buffers, you would describe the schema in the Protocol Buffers interface definition language (IDL) like this:

```
syntax = "proto3";

message Person {
    string user_name = 1;
    int64 favorite_number = 2;
    repeated string interests = 3;
}
```

Protocol Buffers comes with a code generation tool that takes a schema definition like the one shown here, and produces classes that implement the schema in various programming languages. Your application code can call this generated code to encode or decode records of the schema. The schema language is very simple compared to JSON Schema: it only defines the fields of records and their types, but it does not support other restrictions on the possible values of fields.

Encoding Example 5-2 using a Protocol Buffers encoder requires 33 bytes, as shown in Figure 5-3 [14].

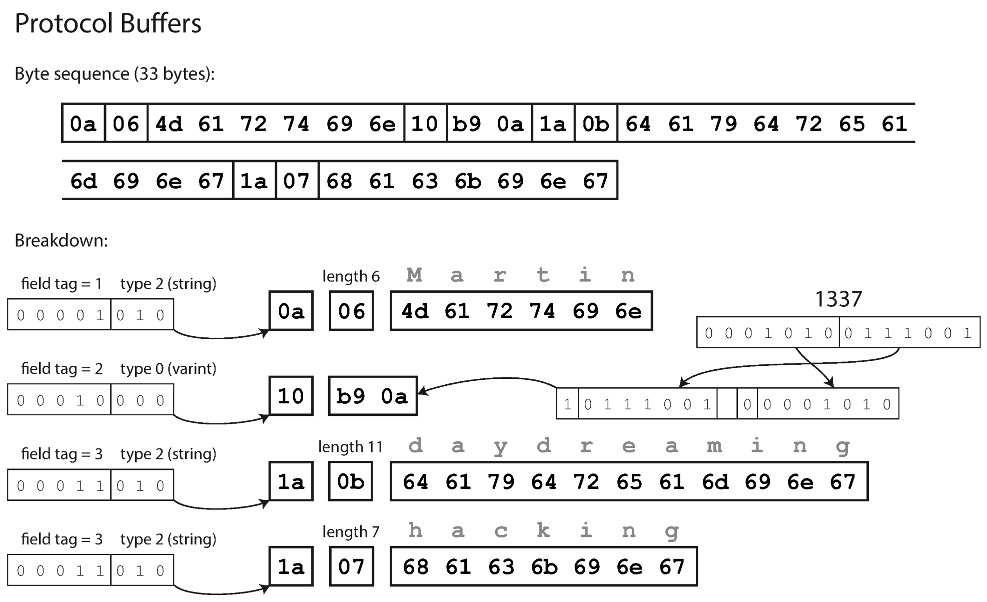


Figure 5-3. Example record encoded using Protocol Buffers.

Similarly to Figure 5-2, each field has a type annotation (to indicate whether it is a string, integer, etc.) and, where required, a length indication (such as the length of a string). The strings that appear in the data ("Martin", "daydreaming", "hacking") are also encoded as ASCII (to be precise, UTF-8), similar to before.

The big difference compared to Figure 5-2 is that there are no field names (`userName` , `favoriteNumber` , `interests`). 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're talking about, without having to spell out the field name.

As you can see, Protocol Buffers saves even more space by packing the field type and tag number into a single byte. It uses variable-length integers: the number 1337 is encoded in two bytes, with the top bit of each byte used to indicate whether there are still more bytes to come (the least significant 7 bits are stored in the first byte to simplify reconstructing the integer as bytes are read). This means numbers between –64 and 63 are encoded in one byte, numbers between –8192 and 8191 are encoded in two bytes, etc. Bigger numbers use more bytes.

Protocol Buffers doesn't have an explicit list or array datatype. Instead, the `repeated` modifier on the `interests` field indicates that the field contains a list of values, rather than a single value. In the binary encoding, the list elements are represented simply as repeated occurrences of the same field tag within the same record.

Field tags and schema evolution

We said previously that schemas inevitably need to change over time. We call this *schema evolution*. How does Protocol Buffers handle schema changes while keeping backward and forward compatibility?

As you can see from the examples, an encoded record is just the concatenation of its encoded fields. Each field is identified by its tag number (the numbers `1`, `2`, `3` in the sample schema) and annotated with a datatype (e.g., string or integer). If a field value is not set, it is simply omitted from the encoded record. From this you can see 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.

You can add new fields to the schema, provided that you give each field a new tag number. If old code (which doesn’t know about the new tag numbers you added) tries to read data written by new code, including a new field with a tag number it doesn’t recognize, it can simply ignore that field. The datatype annotation allows the parser to determine how many bytes it needs to skip, and preserve the unknown fields to avoid the problem in Figure 5-1. This maintains forward compatibility: old code can read records that were written by new code.

What about backward compatibility? 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. If a field was added in the new schema, and you read old data that does not yet contain that field, it is filled in with a default value (for example, the empty string if the field type is string, or zero if it’s a number).

Removing a field is just like adding a field, with backward and forward compatibility concerns reversed. You can never use the same tag number again, because you may still have data written somewhere that includes the old tag number, and that field must be ignored by new code. Tag numbers used in the past can be reserved in the schema definition to ensure they are not forgotten.

What about changing the datatype of a field? That is possible with some types —check the documentation for details—but there is a risk that values will 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.

Avro

Apache Avro is another binary encoding format that is interestingly different from Protocol Buffers. It was started in 2009 as a subproject of Hadoop, as a result of Protocol Buffers not being a good fit for Hadoop's use cases [15].

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. Like Protocol Buffers, this schema language specifies only fields and their types, and not complex validation rules like in JSON Schema.

Our example schema, written in Avro IDL, might look like this:

```
record Person {
    string                userName;
    union { null, long } favoriteNumber = null;
    array<string>         interests;
}
```

The equivalent JSON representation of that schema is as follows:

```
{
    "type": "record",
    "name": "Person",
    "fields": [
        {"name": "userName",       "type": "string"},
        {"name": "favoriteNumber", "type": ["null", "lo
        {"name": "interests",      "type": {"type": "ar
    ]
}
```

First of all, notice that there are no tag numbers in the schema. If we encode our example record (Example 5-2) using this schema, the Avro binary encoding is just 32 bytes long—the most compact of all the encodings we have seen. The breakdown of the encoded byte sequence is shown in Figure 5-4.

If you examine the byte sequence, you can see that there is nothing to identify fields or their datatypes. 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 you 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.

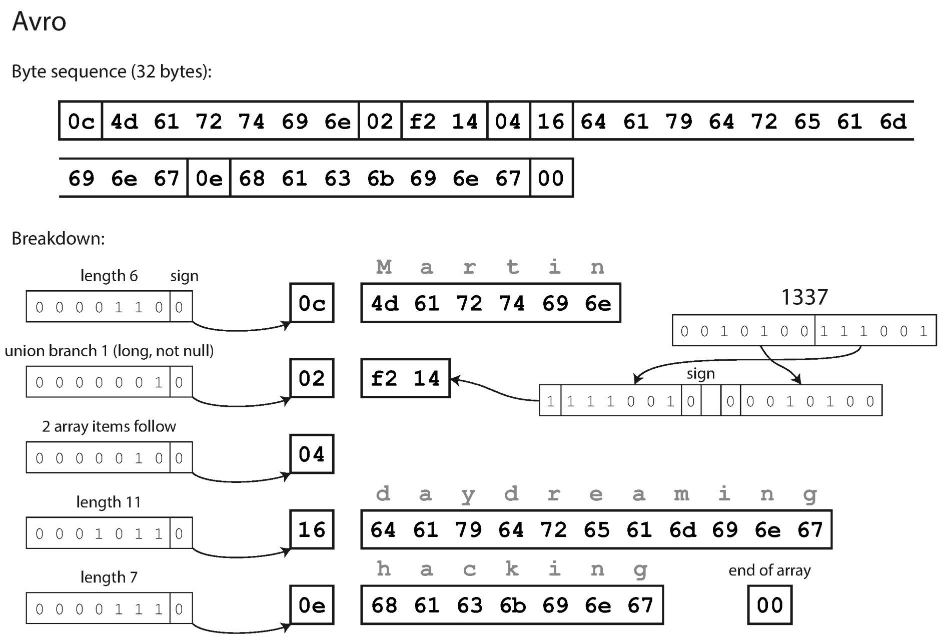


Figure 5-4. Example record encoded using Avro.

To parse the binary data, you 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.

So, how does Avro support schema evolution?

The writer's schema and the reader'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—for example, that schema may be compiled into the application. This is known as the *writer's schema*.

When an application wants to decode some data (read it from a file or database, receive it from the network, etc.), it uses two schemas: the writer's schema that is identical to the one used for encoding, and the *reader's schema*, which may be different. This is illustrated in Figure 5-5. The reader's schema defines the fields of each record that the application code is expecting, and their types.

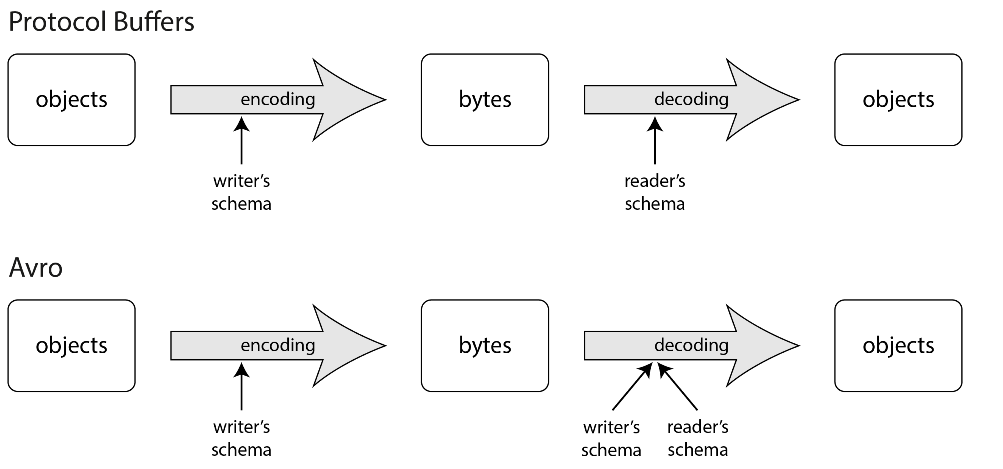
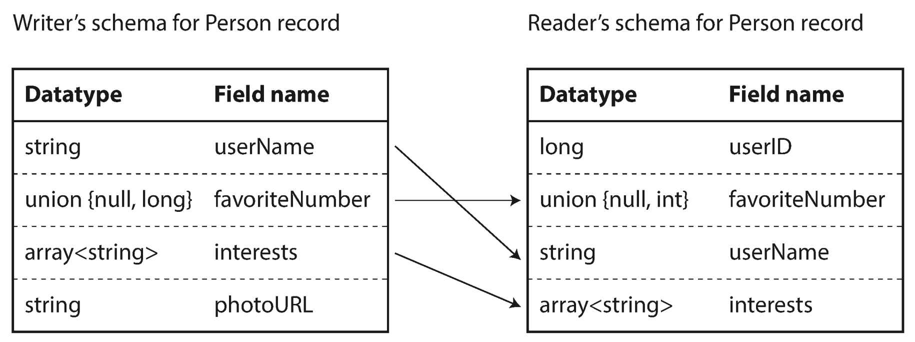


Figure 5-5. In Protocol Buffers, encoding and decoding can use different versions of a schema. In Avro, decoding uses two schemas: the writer's schema must be identical to the one used for encoding, but the reader's schema can be an older or newer version.

If the reader's and writer's schema are the same, decoding is easy. If they are different, Avro 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 [16, 17] defines exactly how this resolution works, and it is illustrated in Figure 5-6.

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.


Writer's schema for Person record
Reader's schema for Person record

Datatype	Field name
string	userName
union {null, long}	favoriteNumber
array<string>	interests
string	photoURL

Datatype	Field name
long	userID
union {null, int}	favoriteNumber
string	userName
array<string>	interests

Figure 5-6. An Avro reader resolves differences between the writer's schema and the reader's schema.

Schema evolution rules

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.

To maintain compatibility, you may only add or remove a field that has a default value. (The field `favoriteNumber` in our Avro schema has a default value of `null`.) For example, say you 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.

If you were to add a field that has no default value, new readers wouldn't be able to read data written by old writers, so you would break backward compatibility. If you were to remove a field that has no default value, old readers wouldn't be able to read data written by new writers, so you would break forward compatibility.

In some programming languages, `null` is an acceptable default for any variable, but this is not the case in Avro: if you want to allow a field to be null, you have to use a *union type*. For example, `union { null, long, string } field;` indicates that `field` can be a number, or a string, or null. You can only use `null` as a default value if it is the first branch of the union. This is a little more verbose than having everything nullable by default, but it helps prevent bugs by being explicit about what can and cannot be null [18].

Changing the datatype of a field is possible, provided that Avro can convert the type. Changing the name of a field is possible but a little tricky: 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.

But what is the writer’s schema?

There is an important question that we’ve glossed over so far: how does the reader know the writer’s schema with which a particular piece of data was encoded? We can’t just include the entire schema with every record, because the schema would likely be much bigger than the encoded data, making all the space savings from the binary encoding futile.

The answer depends on the context in which Avro is being used. To give a few examples:

Large file with lots of records

A common use for Avro is for storing a large file containing millions of records, all encoded with the same schema. (We will discuss this kind of situation in Chapter 11.) In this case, the writer of that file can just include the writer’s schema once at the beginning of the file. Avro specifies a file format (object container files) to do this.

Database with individually written records

In a database, different records may be written at different points in time using different writer’s schemas—you cannot assume that all the records will have the same schema. The simplest solution is to include a version number at the beginning of every encoded record, and to keep a list of schema versions in your database. A reader can fetch a record, extract the version number, and then fetch the writer’s schema for that version number from the database. Using that writer’s schema, it can decode the rest of the record. Confluent’s schema registry for Apache Kafka [19] and LinkedIn’s Espresso [20] work this way, for example.

Sending records over a network connection

When two processes are communicating over a bidirectional network connection, they can negotiate the schema version on connection setup and then use that schema for the lifetime of the connection. The Avro

RPC protocol (see “Dataflow Through Services: REST and RPC”) works like this.

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 [21]. As the version number, you could use a simple incrementing integer, or you could use a hash of the schema.

Dynamically generated schemas

One advantage of Avro’s approach, compared to Protocol Buffers, is that the schema doesn’t contain any tag numbers. But why is this important? What’s the problem with keeping a couple of numbers in the schema?

The difference is that Avro is friendlier to *dynamically generated* schemas. For example, say you have a relational database whose contents you want to dump to a file, and you want to use a binary format to avoid the aforementioned problems with textual formats (JSON, CSV, XML). If you use Avro, you can fairly easily generate an Avro schema (in the JSON representation we saw earlier) from the relational schema and encode the database contents using that schema, dumping it all to an Avro object container file [22]. You can generate a record schema for each database table, and each column becomes a field in that record. The column name in the database maps to the field name in Avro.

Now, if the database schema changes (for example, a table has one column added and one column removed), you can just generate a new Avro schema from the updated database schema and export data in the new Avro schema. The data export process does not need to pay any attention to the schema change—it can simply do the schema conversion every time it runs. Anyone who reads the new data files will see that the fields of the record have changed, but since the fields are identified by name, the updated writer’s schema can still be matched up with the old reader’s schema.

By contrast, if you were using Protocol Buffers for this purpose, the field tags would likely have to be assigned by hand: every time the database schema changes, an administrator would have to manually update the mapping from database column names to field tags. (It might be possible to automate this, but the schema generator would have to be very careful to not assign previously used field tags.) This kind of dynamically generated schema simply wasn't a design goal of Protocol Buffers, whereas it was for Avro.

The Merits of Schemas

As we saw, Protocol Buffers and Avro both use a schema to describe a binary encoding format. Their schema languages are much simpler than XML Schema or JSON Schema, which support much more detailed validation rules (e.g., "the string value of this field must match this regular expression" or "the integer value of this field must be between 0 and 100"). As Protocol Buffers and Avro are simpler to implement and simpler to use, they have grown to support a fairly wide range of programming languages.

The ideas on which these encodings are based are by no means new. For example, they have a lot in common with ASN.1, a schema definition language that was first standardized in 1984 [23, 24]. It was used to define various network protocols, and its binary encoding (DER) is still used to encode SSL certificates (X.509), for example [25]. ASN.1 supports schema evolution using tag numbers, similar to Protocol Buffers [26]. However, it's also very complex and badly documented, so ASN.1 is probably not a good choice for new applications.

Many data systems also implement some kind of proprietary binary encoding for their data. For example, most relational databases have a network protocol over which you can send queries to the database and get back responses. Those protocols are generally specific to a particular database, and the database vendor provides a driver (e.g., using the ODBC or JDBC APIs) that decodes responses from the database's network protocol into in-memory data structures.

So, we can see that although textual data formats such as JSON, XML, and CSV are widespread, binary encodings based on schemas are also a viable option. They have a number of nice properties:

- They can be much more compact than the various "binary JSON" variants, since they can omit field names from the encoded data.
- The schema is a valuable form of documentation, and because the schema is required for decoding, you can be sure that it is up to date (whereas manually maintained documentation may easily diverge from reality).

- Keeping a database of schemas allows you to check forward and backward compatibility of schema changes, before anything is deployed.
- For users of statically typed programming languages, the ability to generate code from the schema is useful, since it enables type-checking at compile time.

In summary, schema evolution allows the same kind of flexibility as schemaless/schema-on-read JSON databases provide (see “Schema flexibility in the document model”), while also providing better guarantees about your data and better tooling. Still, it’s advisable to keep the number of concurrent schema formats to a minimum to keep operations simple.

Modes of Dataflow

At the beginning of this chapter we said that whenever you want to send some data to another process with which you don’t share memory—for example, whenever you want to send data over the network or write it to a file—you need to encode it as a sequence of bytes. We then discussed a variety of different encodings for doing this.

We talked about forward and backward compatibility, which are important for evolvability (making change easy by allowing you to upgrade different parts of your system independently, and not having to change everything at once). Compatibility is a relationship between one process that encodes the data, and another process that decodes it.

That’s a fairly abstract idea—there are many ways data can flow from one process to another. Who encodes the data, and who decodes it? In the rest of this chapter we will explore some of the most common ways how data flows between processes:

- Via databases (see “Dataflow Through Databases”)
- Via service calls (see “Dataflow Through Services: REST and RPC”)
- Via workflow engines (see “Durable Execution and Workflows”)
- Via asynchronous messages (see “Event-Driven Architectures”)

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.

In general, it's common for several different processes to be accessing a database at the same time. Those processes might be several different applications or services, or they may simply be several instances of the same service (running in parallel for scalability or fault tolerance). Either way, in an environment where the application is changing, it is likely that some processes accessing the database will be running newer code and some will be running older code—for example because a new version is currently being deployed in a rolling upgrade, so some instances have been updated while others haven't yet.

This means that a value in the database may be written by a *newer* version of the code, and subsequently read by an *older* version of the code that is still running. Thus, forward compatibility is also often required for databases.

Different values written at different times

A database generally allows any value to be updated at any time. This means that within a single database you may have some values that were written five milliseconds ago, and some values that were written five years ago.

When you deploy a new version of your application (of a server-side application, at least), you may entirely replace the old version with the new version within a few minutes. The same is not true of database contents: the five-year-old data will still be there, in the original encoding, unless you have explicitly rewritten it since then. This observation is sometimes summed up as *data outlives code*.

Rewriting (*migrating*) data into a new schema is certainly possible, but it's an expensive thing to do on a large dataset, so most databases defer the operation to be asynchronous and best-effort. For example, LSM tree storage engines (see “Log-Structured Storage”) will rewrite data using the latest format during compaction. Most relational databases also allow simple schema changes, such as adding a new column with a `null` default value, without rewriting existing data. When an old row is read, the database fills in `null` s for any columns that are missing from the encoded data on disk. Schema evolution thus allows the entire database to appear as if it was encoded with a single schema, even though the underlying storage may contain records encoded with various historical versions of the schema.

More complex schema changes—for example, changing a single-valued attribute to be multi-valued, or moving some data into a separate table—still require data to be rewritten, often at the application level [27]. Maintaining forward and backward compatibility across such migrations is still a research problem [28].

Archival storage

Perhaps you take a snapshot of your database from time to time, say for backup purposes or for loading into a data warehouse (see “Data Warehousing”). In this case, the data dump will typically be encoded using the latest schema, even if the original encoding in the source database contained a mixture of schema versions from different eras. Since you’re copying the data anyway, you might as well encode the copy of the data consistently.

As the data dump is written in one go and is thereafter immutable, formats like Avro object container files are a good fit. This is also a good opportunity to encode the data in an analytics-friendly column-oriented format such as Parquet (see “Column Compression”).

In Chapter 11 we will talk more about using data in archival storage.

Dataflow Through Services: REST and RPC

When you have processes that need to communicate over a network, there are a few different ways of arranging that communication. The most common arrangement is to have two roles: *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*.

The web works this way: clients (web browsers) make requests to web servers, making `GET` requests to download HTML, CSS, JavaScript, images, etc., and making `POST` requests to submit data to the server. The API consists of a standardized set of protocols and data formats (HTTP, URLs, SSL/TLS, HTML, etc.). Because web browsers, web servers, and website authors mostly agree on these standards, you can use any web browser to access any website (at least in theory!).

Web browsers are not the only type of client. For example, native apps running on mobile devices and desktop computers often talk to servers, and client-side JavaScript applications running inside web browsers can also make HTTP requests. In this case, the server's response is typically not HTML for displaying to a human, but rather data in an encoding that is convenient for further processing by the client-side application code (most often JSON). Although HTTP may be used as the transport protocol, the API implemented on top is application-specific, and the client and server need to agree on the details of that API.

In some ways, services are similar to databases: they typically allow clients to submit and query data. However, while databases allow arbitrary queries using the query languages we discussed in Chapter 3, services expose an application-specific API that only allows inputs and outputs that are predetermined by the business logic (application code) of the service [29]. This restriction provides a degree of encapsulation: services can impose fine-grained restrictions on what clients can and cannot do.

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. A common principle is that 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. We should therefore 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. But as long as APIs remain compatible, teams are free to modify their systems in any way they'd like; this property makes it much easier for developers to do internal migrationes of data, services, or even entire systems.

Web services

When HTTP is used as the underlying protocol for talking to the service, it is called a *web service*. Web services are commonly used when building a service oriented or microservices architecture (discussed earlier in "Microservices and Serverless"). The term "web service" is perhaps a slight misnomer, because web services are not only used on the web, but in several different contexts. For example:

1. A client application running on a user's device (e.g., a native app on a mobile device, or a JavaScript web app in a browser) 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 private network, as part of a service-oriented/microservices architecture.
3. One service making requests to a service owned by a different organization, usually via the internet. This is used for data exchange between different organizations' 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.

The most popular service design philosophy is REST, which builds upon the principles of HTTP [30, 31]. It emphasizes simple data formats, using URLs for identifying resources and using HTTP features for cache control, authentication, and content type negotiation. An API designed according to the principles of REST is called *RESTful*.

Code that needs to invoke a web service API must know which HTTP endpoint to query, and what data format to send and expect in response. Even if a service adopts RESTful design principles, clients need to somehow find out these details. Service developers often use an interface definition language (IDL) to define and document their service's API endpoints and data models, and to evolve them over time. Other developers can then use the service definition to determine how to query the service. The two most popular service IDLs are OpenAPI (also known as Swagger [32]) and gRPC. OpenAPI is used for web services that send and receive JSON data, while gRPC services send and receive Protocol Buffers.

Developers typically write OpenAPI service definitions in JSON or YAML; see Example 5-3. The service definition allows developers to define service endpoints, documentation, versions, data models, and much more. gRPC definitions look similar, but are defined using Protocol Buffers service definitions.

Example 5-3. Example OpenAPI service definition in YAML

```yaml
openapi: 3.0.0
info:
  title: Ping, Pong
  version: 1.0.0
servers:
  - url: http://localhost:8080
paths:
  /ping:
    get:
      summary: Given a ping, returns a pong message
      responses:
        '200':
          description: A pong
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
                    example: Pong!
```

Even if a design philosophy and IDL are adopted, developers must still write the code that implements their service's API calls. A service framework is often adopted to simplify this effort. Service frameworks such as Spring Boot, FastAPI, and gRPC allow developers to write the business logic for each API endpoint while the framework code handles routing, metrics, caching, authentication, and so on. Example 5-4 shows an example Python implementation of the service defined in Example 5-3.

Example 5-4. Example FastAPI service implementing the definition from Example 5-3

```python
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI(title="Ping, Pong", version="1.0.0")
```

```
class PongResponse(BaseModel):
    message: str = "Pong!"

@app.get("/ping", response_model=PongResponse,
         summary="Given a ping, returns a pong message'
async def ping():
    return PongResponse()
```

Many frameworks couple service definitions and server code together. In some cases, such as with the popular Python FastAPI framework, servers are written in code and an IDL is generated automatically. In other cases, such as with gRPC, the service definition is written first, and server code scaffolding is generated. Both approaches allow developers to generate client libraries and SDKs in a variety of languages from the service definition. In addition to code generation, IDL tools such as Swagger's can generate documentation, verify schema change compatibility, and provide a graphical user interfaces for developers to query and test services.

The problems with remote procedure calls (RPCs)

Web services are merely the latest incarnation of a long line of technologies for making API requests over a network, many of which received a lot of hype but have serious problems. Enterprise JavaBeans (EJB) and Java's Remote Method Invocation (RMI) are limited to Java. The Distributed Component Object Model (DCOM) is limited to Microsoft platforms. The Common Object Request Broker Architecture (CORBA) is excessively complex, and does not provide backward or forward compatibility [33]. SOAP and the WS-* web services framework aim to provide interoperability across vendors, but are also plagued by complexity and compatibility problems [34, 35, 36].

All of these are based on the idea of a *remote procedure call* (RPC), which has been around since the 1970s [37]. 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, within the same process (this abstraction is called *location transparency*). Although RPC seems convenient at first, the approach is fundamentally flawed [38, 39]. A network request is very different from a local function call:

- A local function call is predictable and either succeeds or fails, depending only on parameters that are under your control. A network request is unpredictable: the request or response may be lost due to a network problem, or the remote machine may be slow or unavailable, and such problems are entirely outside of your control. Network problems are common, so you have to anticipate them, for example by retrying a failed request.

- A local function call either returns a result, or throws an exception, or never returns (because it goes into an infinite loop or the process crashes). A network request has another possible outcome: it may return without a result, due to a *timeout*. In that case, you simply don't know what happened: if you don't get a response from the remote service, you have no way of knowing whether the request got through or not. (We discuss this issue in more detail in Chapter 9.)
- If you retry a failed network request, it could happen that the previous request actually got through, and only the response was lost. In that case, retrying will cause the action to be performed multiple times, unless you build a mechanism for deduplication (*idempotence*) into the protocol [40]. Local function calls don't have this problem. (We discuss idempotence in more detail in Chapter 12.)
- Every time you call a local function, it normally takes about the same time to execute. A network request is much slower than a function call, and its latency is also wildly variable: at good times it may complete in less than a millisecond, but when the network is congested or the remote service is overloaded it may take many seconds to do exactly the same thing.
- When you call a local function, you can efficiently pass it references (pointers) to objects in local memory. When you make a network request, all those parameters need to be encoded into a sequence of bytes that can be sent over the network. That's okay if the parameters are immutable primitives like numbers or short strings, but it quickly becomes problematic with larger amounts of data and mutable objects.
- The client and the service may be implemented in different programming languages, so the RPC framework must translate datatypes from one language into another. This can end up ugly, since not all languages have the same types—recall JavaScript's problems with numbers greater than 2^{53}, for example (see "JSON, XML, and Binary Variants"). This problem doesn't exist in a single process written in a single language.

All of these factors mean that there's no point trying to make a remote service look too much like a local object in your programming language, because it's a fundamentally different thing. Part of the appeal of REST is that it treats state transfer over a network as a process that is distinct from a function call.

Load balancers, service discovery, and service meshes

All services communicate over the network. For this reason, a client must know the address of the service it's connecting to—a problem known as *service discovery*. The simplest approach is to configure a client to connect to the IP address and port where the service is running. This configuration will work, but if the server goes offline, is transferred to a new machine, or becomes overloaded, the client has to be manually reconfigured.

To provide higher availability and scalability, there are usually multiple instances of a service running on different machines, any of which can handle an incoming request. Spreading requests across these instances is called *load balancing* [41]. There are many load balancing and service discovery solutions available:

- *Hardware load balancers* are specialized pieces of equipment that are installed in data centers. They allow clients to connect to a single host and port, and incoming connections are routed to one of the servers running the service. Such load balancers detect network failures when connecting to a downstream server and shift the traffic to other servers.
- *Software load balancers* behave in much the same way as hardware load balancers. But rather than requiring a special appliance, software load balancers such as Nginx and HAProxy are applications that can be installed on a standard machine.
- The *domain name service (DNS)* is how domain names are resolved on the Internet when you open a webpage. It supports load balancing by allowing multiple IP addresses to be associated with a single domain name. Clients can then be configured to connect to a service using a domain name rather than IP address, and the client's network layer picks which IP address to use when making a connection. One drawback of this approach is that DNS is designed to propagate changes over longer periods of time, and to cache DNS entries. If servers are started, stopped, or moved frequently, clients might see stale IP addresses that no longer have a server running on them.
- *Service discovery systems* use a centralized registry such as etcd or Apache ZooKeeper rather than DNS to track which service endpoints are available (we'll return to these systems in "Coordination Services"). When a new

service instance starts up, it registers itself with the service discovery system by declaring the host and port it's listening on, along with relevant metadata such as shard ownership information (see Chapter 7), data center location, and more. The service then periodically sends a heartbeat signal to the discovery system to signal that the service is still available.

  When a client wishes to connect to a service, it first queries the discovery system to get a list of available endpoints, and then connects directly to the endpoint. Compared to DNS, service discovery supports a much more dynamic environment where service instances change frequently. Discovery systems also give clients more metadata about the service they're connecting to, which enables clients to make smarter load balancing decisions.

- *Service meshes* are a sophisticated form of load balancing that combine software load balancers and service discovery. Unlike traditional software load balancers, which run on a separate machine, service mesh load balancers are typically deployed as an in-process client library or as a process or "sidecar" container on both the client and server. Client applications connect to their own local service load balancer, which connects to the server's load balancer. From there, the connection is routed to the local server process.

  Though complicated, this topology offers a number of advantages. Because the clients and servers are routed entirely through local connections, connection encryption can be handled entirely at the load balancer level. This shields clients and servers from having to deal with the complexities of SSL certificates and TLS. Mesh systems also provide sophisticated observability. They can track which services are calling each other in realtime, detect failures, track traffic load, and more.

Which solution is appropriate depends on an organization's needs. Those running in a very dynamic service environment with an orchestrator such as Kubernetes often choose to run a service mesh such as Istio or Linkerd. Specialized infrastructure such as databases or messaging systems might require their own purpose-built load balancer. Simpler deployments are best served with software load balancers.

Data encoding and evolution for RPC

For evolvability, it is important that RPC clients and servers can be changed and deployed independently. Compared to data flowing through databases (as described in the last section), we can make a simplifying assumption in the case of dataflow through services: it is reasonable to assume that all the servers will be updated first, and all the clients second. Thus, you only need backward compatibility on requests, and forward compatibility on responses.

The backward and forward compatibility properties of an RPC scheme are inherited from whatever encoding it uses:

- gRPC (Protocol Buffers) and Avro RPC can be evolved according to the compatibility rules of the respective encoding format.
- RESTful APIs most commonly use JSON for responses, and JSON or URI-encoded/form-encoded request parameters for requests. Adding optional request parameters and adding new fields to response objects are usually considered changes that maintain compatibility.

Service compatibility is made harder by the fact that RPC is often used for communication across organizational boundaries, so the provider of a service often has no control over its clients and cannot force them to upgrade. Thus, compatibility needs to be maintained for a long time, perhaps indefinitely. If a compatibility-breaking change is required, the service provider often ends up maintaining multiple versions of the service API side by side.

There is no agreement on how API versioning should work (i.e., how a client can indicate which version of the API it wants to use [42]). For RESTful APIs, common approaches are to use a version number in the URL or in the HTTP `Accept` header. For services that use API keys to identify a particular client, another option is to store a client's requested API version on the server and to allow this version selection to be updated through a separate administrative interface [43].

Durable Execution and Workflows

By definition, service-based architectures have multiple services that are all responsible for different portions of an application. Consider a payment processing application that charges a credit card and deposits the funds into a bank account. This system would likely have different services responsible for fraud detection, credit card integration, bank integration, and so on.

Processing a single payment in our example requires many service calls. A payment processor service might invoke the fraud detection service to check for fraud, call the credit card service to debit the credit card, and call the banking service to deposit debited funds, as shown in Figure 5-7. We call this sequence of steps a *workflow*, and each step a *task*. Workflows are typically defined as a graph of tasks. Workflow definitions may be written in a general-purpose programming language, a domain specific language (DSL), or a markup language such as Business Process Execution Language (BPEL) [44].

TASKS, ACTIVITIES, AND FUNCTIONS

Different workflow engines use different names for tasks. Temporal, for example, uses the term *activity*. Others refer to tasks as *durable functions*. Though the names differ, the concepts are the same.

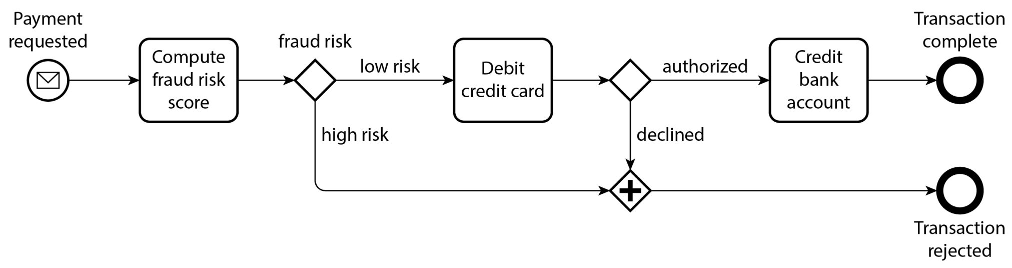


Figure 5-7. Example of a workflow expressed using Business Process Model and Notation (BPMN), a graphical notation.

Workflows are run, or executed, by a *workflow engine*. Workflow engines determine when to run each task, on which machine a task must be run, what to do if a task fails (e.g., if the machine crashes while the task is running), how many tasks are allowed to execute in parallel, and more.

Workflow engines are typically composed of an orchestrator and an executor. The orchestrator is responsible for scheduling tasks to be executed and the executor is responsible for executing tasks. Execution begins when a workflow is triggered. The orchestrator triggers the workflow itself if users define a time-based schedule, such as hourly execution. External sources such as a web service or even a human can also trigger workflow executions. Once triggered, executors are invoked to run tasks.

There are many kinds of workflow engines that address a diverse set of use cases. Some, such as Airflow, Dagster, and Prefect, integrate with data systems and orchestrate ETL tasks. Others, such as Camunda and Orkes, provide a graphical notation for workflows (such as BPMN, used in Figure 5-7) so that non-engineers can more easily define and execute workflows. Still others, such as Temporal and Restate provide *durable execution*.

Durable execution

Durable execution frameworks have become a popular way to build service-based architectures that require transactionality. In our payment example, we would like to process each payment exactly once. A failure while the workflow is executing could result in a credit card charge, but no corresponding bank account deposit. In a service-based architecture, we can't simply wrap the two tasks in a database transaction. Moreover, we might be interacting with third-party payment gateways that we have limited control over.

Durable execution frameworks are a way to provide *exactly-once semantics* for workflows. If a task fails, the framework will re-execute the task, but will skip any RPC calls or state changes that the task made successfully before failing. Instead, the framework will pretend to make the call, but will instead return the results from the previous call. This is possible because durable execution frameworks log all RPCs and state changes to durable storage like a write-ahead log [45, 46]. Example 5-5 shows an example of a workflow definition that supports durable execution using Temporal.

Example 5-5. A Temporal workflow definition fragment for the payment workflow in Figure 5-7.

```
@workflow.defn
class PaymentWorkflow:
    @workflow.run
    async def run(self, payment: PaymentRequest) -> Pay
        is_fraud = await workflow.execute_activity(
            check_fraud,
            payment,
            start_to_close_timeout=timedelta(seconds=15
        )
        if is_fraud:
            return PaymentResultFraudulent
        credit_card_response = await workflow.execute_a
            debit_credit_card,
            payment,
            start_to_close_timeout=timedelta(seconds=15
        )
        # ...
```

Frameworks like Temporal are not without their challenges. External services, such as the third-party payment gateway in our example, must still provide an idempotent API. Developers must remember to use unique IDs for these APIs to prevent duplicate execution [47]. And because durable execution frameworks log each RPC call in order, it expects a subsequent execution to make the same RPC calls in the same order. This makes code changes brittle: you might introduce undefined behavior simply by re-ordering function calls [48]. Instead of modifying the code of an existing workflow, it is safer to deploy a new version of the code separately, so that re-executions of existing workflow invocations continue to use the old version, and only new invocations use the new code [49].

Similarly, because durable execution frameworks expect to replay all code deterministically (the same inputs produce the same outputs), nondeterministic code such as random number generators or system clocks are problematic [48]. Frameworks often provide their own, deterministic implementations of such library functions, but you have to remember to use them. In some cases, such as with Temporal's workflowcheck tool, frameworks provide static analysis tools to determine if nondeterministic behavior has been introduced.

NOTE

Making code deterministic is a powerful idea, but tricky to do robustly. In "The Power of Determinism" we will return to this topic.

Event-Driven Architectures

In this final section, we will briefly look at *event-driven architectures*, which are another way how encoded data can flow from one process to another. A request is called an *event* or *message*; unlike RPC, the sender usually does not wait for the recipient to process the event. Moreover, events are typically not sent to the recipient via a direct network connection, but go via an intermediary called a *message broker* (also called an *event broker*, *message queue*, or *message-oriented middleware*), which stores the message temporarily. [50].

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 need for service discovery, since senders do not need to directly connect to the IP address of the recipient.
- It allows the same 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 via a message broker is *asynchronous*: the sender doesn't wait for the message to be delivered, but simply sends it and then forgets about it. It's possible to implement a synchronous RPC-like model by having the sender wait for a response on a separate channel.

Message brokers

In the past, the landscape of message brokers was dominated by commercial enterprise software from companies such as TIBCO, IBM WebSphere, and webMethods, before open source implementations such as RabbitMQ, ActiveMQ, HornetQ, NATS, Redpanda, and Apache Kafka become popular. More recently, cloud services such as Amazon Kinesis, Azure Service Bus, and Google Cloud Pub/Sub have gained adoption. We will compare them in more detail in Chapter 12.

The detailed delivery semantics vary by implementation and configuration, but in general, two message distribution patterns are most often used:

- One process adds a message to a named *queue*, and a *consumer* of the queue then receives the message. If there are multiple consumers, one of them receives the message.
- One process publishes a message to a named *topic*, and the broker delivers that message to all *subscribers* of that topic. If there are multiple subscribers, they all receive the message.

Message brokers typically don't enforce any particular data model—a message is just a sequence of bytes with some metadata, so you can use any encoding format. A common approach is to use Protocol Buffers, Avro, or JSON, and to deploy a schema registry alongside the message broker to store all the valid schema versions and check their compatibility [19, 21].

AsyncAPI, a messaging-based equivalent of OpenAPI, can also be used to specify the schema of messages.

Message brokers differ in terms of how durable their messages are. Many write messages to disk, so that they are not lost in case the message broker crashes or needs to be restarted. Unlike databases, many message brokers automatically delete messages again after they have been consumed. Some brokers can be configured to store messages indefinitely, which you would require if you want to use event sourcing (see "Event Sourcing and CQRS").

If a consumer republishes messages to another topic, you may need to be careful to preserve unknown fields, to prevent the issue described previously in the context of databases (Figure 5-1).

Distributed actor frameworks

The *actor model* is a programming model for concurrency in a single process. Rather than dealing directly with threads (and the associated problems of race conditions, locking, and deadlock), logic is encapsulated in *actors*. Each actor typically represents one client or entity, it may have some local state (which is not shared with any other actor), and it communicates with other actors by sending and receiving asynchronous messages. Message delivery is not guaranteed: in certain error scenarios, messages will be lost. Since each actor processes only one message at a time, it doesn't need to worry about threads, and each actor can be scheduled independently by the framework.

In *distributed actor frameworks* such as Akka, Orleans [51], and Erlang/OTP, this programming model is used to scale an application across multiple nodes. The same message-passing mechanism is used, no matter whether the sender and recipient are on the same node or different nodes. If they are on different nodes, the message is transparently encoded into a byte sequence, sent over the network, and decoded on the other side.

Location transparency works better in the actor model than in RPC, because the actor model already assumes that messages may be lost, even within a single process. Although latency over the network is likely higher than within the same process, there is less of a fundamental mismatch between local and remote communication when using the actor model.

A distributed actor framework essentially integrates a message broker and the actor programming model into a single framework. However, if you want to perform rolling upgrades of your actor-based application, you still have to worry about forward and backward compatibility, as messages may be sent from a node running the new version to a node running the old version, and vice versa. This can be achieved by using one of the encodings discussed in this chapter.

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 evolving them.

In particular, 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 (thus encouraging frequent small releases over rare big releases) 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 (new code can read old data) and forward compatibility (old code can read new data).

We discussed several data encoding formats and their compatibility properties:

- Programming language–specific encodings are restricted to a single programming language and often fail to provide forward and backward compatibility.
- Textual formats like JSON, XML, and CSV are widespread, and their compatibility depends on how you use them. They have optional schema languages, which are sometimes helpful and sometimes a hindrance. These

formats are somewhat vague about datatypes, so you have to be careful with things like numbers and binary strings.
- Binary schema–driven formats like 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, these formats have the downside that data needs to be decoded before it is human-readable.

We also discussed several modes of dataflow, illustrating different scenarios in which data encodings are important:

- Databases, where the process writing to the database encodes the data and the process reading from the database decodes it
- RPC and REST APIs, where the client encodes a request, the server decodes the request and encodes a response, and the client finally decodes the response
- Event-driven architectures (using message brokers or actors), where nodes communicate by sending each other messages that are encoded by the sender and decoded by the recipient

We can conclude that with a bit of care, backward/forward compatibility and rolling upgrades are quite achievable. May your application's evolution be rapid and your deployments be frequent.

FOOTNOTES

REFERENCES

[1] CWE-502: Deserialization of Untrusted Data. Common Weakness Enumeration, *cwe.mitre.org*, July 2006. Archived at perma.cc/26EU-UK9Y

[2] Steve Breen. What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability. *foxglovesecurity.com*, November 2015. Archived at perma.cc/9U97-UVVD

[3] Patrick McKenzie. What the Rails Security Issue Means for Your Startup. *kalzumeus.com*, January 2013. Archived at perma.cc/2MBJ-7PZ6

[4] Brian Goetz. Towards Better Serialization. *openjdk.org*, June 2019. Archived at perma.cc/UK6U-GQDE

[5] Eishay Smith. jvm-serializers wiki. *github.com*, October 2023. Archived at perma.cc/PJP7-WCNG

[6] XML Is a Poor Copy of S-Expressions. *wiki.c2.com*, May 2013. Archived at perma.cc/7FAN-YBKL

[7] Julia Evans. Examples of floating point problems. *jvns.ca*, January 2023. Archived at perma.cc/M57L-QKKW

[8] Matt Harris. Snowflake: An Update and Some Very Important Information. Email to *Twitter Development Talk* mailing list, October 2010. Archived at perma.cc/8UBV-MZ3D

[9] Yakov Shafranovich. RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files. IETF, October 2005.

[10] Andy Coates. Evolving JSON Schemas - Part I and Part II. *creekservice.org*, January 2024. Archived at perma.cc/MZW3-UA54 and perma.cc/GT5H-WKZ5

[11] Pierre Genevès, Nabil Layaïda, and Vincent Quint. Ensuring Query Compatibility with Evolving XML Schemas. INRIA Technical Report 6711, November 2008.

[12] Tim Bray. Bits On the Wire. *tbray.org*, November 2019. Archived at perma.cc/3BT3-BQU3

[13] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. Facebook technical report, April 2007. Archived at perma.cc/22BS-TUFB

[14] Martin Kleppmann. Schema Evolution in Avro, Protocol Buffers and Thrift. *martin.kleppmann.com*, December 2012. Archived at perma.cc/E4R2-9RJT

[15] Doug Cutting, Chad Walters, Jim Kellerman, et al. [PROPOSAL] New Subproject: Avro. Email thread on *hadoop-general* mailing list, *lists.apache.org*, April 2009. Archived at perma.cc/4A79-BMEB

[16] Apache Software Foundation. Apache Avro 1.12.0 Specification. *avro.apache.org*, August 2024. Archived at perma.cc/C36P-5EBQ

[17] Apache Software Foundation. Avro schemas as LL(1) CFG definitions. *avro.apache.org*, August 2024. Archived at perma.cc/JB44-EM9Q

[18] Tony Hoare. Null References: The Billion Dollar Mistake. Talk at *QCon London*, March 2009.

[19] Confluent, Inc. Schema Registry Overview. *docs.confluent.io*, 2024. Archived at perma.cc/92C3-A9JA

[20] Aditya Auradkar and Tom Quiggle. Introducing Espresso—LinkedIn's Hot New Distributed Document Store. *engineering.linkedin.com*, January 2015. Archived at perma.cc/FX4P-VW9T

[21] Jay Kreps. Putting Apache Kafka to Use: A Practical Guide to Building a Stream Data Platform (Part 2). *confluent.io*, February 2015. Archived at perma.cc/8UA4-ZS5S

[22] Gwen Shapira. The Problem of Managing Schemas. *oreilly.com*, November 2014. Archived at perma.cc/BY8Q-RYV3

[23] John Larmouth. *ASN.1 Complete*. Morgan Kaufmann, 1999. ISBN: 978-0-122-33435-1. Archived at perma.cc/GB7Y-XSXQ

[24] Burton S. Kaliski Jr. A Layman's Guide to a Subset of ASN.1, BER, and DER. Technical Note, RSA Data Security, Inc., November 1993. Archived at perma.cc/2LMN-W9U8

[25] Jacob Hoffman-Andrews. A Warm Welcome to ASN.1 and DER. *letsencrypt.org*, April 2020. Archived at perma.cc/CYT2-GPQ8

[26] Lev Walkin. Question: Extensibility and Dropping Fields. *lionet.info*, September 2010. Archived at perma.cc/VX8E-NLH3

[27] Jacqueline Xu. Online migrations at scale. *stripe.com*, February 2017. Archived at perma.cc/X59W-DK7Y

[28] Geoffrey Litt, Peter van Hardenberg, and Orion Henry. Project Cambria: Translate your data with lenses. Technical Report, *Ink & Switch*, October 2020. Archived at perma.cc/WA4V-VKDB

[29] Pat Helland. Data on the Outside Versus Data on the Inside. At *2nd Biennial Conference on Innovative Data Systems Research* (CIDR), January 2005.

[30] Roy Thomas Fielding. Architectural Styles and the Design of Network-Based Software Architectures. PhD Thesis, University of California, Irvine, 2000. Archived at perma.cc/LWY9-7BPE

[31] Roy Thomas Fielding. REST APIs must be hypertext-driven." *roy.gbiv.com*, October 2008. Archived at perma.cc/M2ZW-8ATG

[32] OpenAPI Specification Version 3.1.0. *swagger.io*, February 2021. Archived at perma.cc/3S6S-K5M4

[33] Michi Henning. The Rise and Fall of CORBA. *Communications of the ACM*, volume 51, issue 8, pages 52–57, August 2008. doi:10.1145/1378704.1378718

[34] Pete Lacey. The S Stands for Simple. *harmful.cat-v.org*, November 2006. Archived at perma.cc/4PMK-Z9X7

[35] Stefan Tilkov. Interview: Pete Lacey Criticizes Web Services. *infoq.com*, December 2006. Archived at perma.cc/JWF4-XY3P

[36] Tim Bray. The Loyal WS-Opposition. *tbray.org*, September 2004. Archived at perma.cc/J5Q8-69Q2

[37] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems* (TOCS), volume 2, issue 1, pages 39–59, February 1984. doi:10.1145/2080.357392

[38] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A Note on Distributed Computing. Sun Microsystems Laboratories, Inc., Technical Report TR-94-29, November 1994. Archived at perma.cc/8LRZ-BSZR

[39] Steve Vinoski. Convenience over Correctness. *IEEE Internet Computing*, volume 12, issue 4, pages 89–92, July 2008. doi:10.1109/MIC.2008.75

[40] Brandur Leach. Designing robust and predictable APIs with idempotency. *stripe.com*, February 2017. Archived at perma.cc/JD22-XZQT

[41] Sam Rose. Load Balancing. *samwho.dev*, April 2023. Archived at perma.cc/Q7BA-9AE2

[42] Troy Hunt. Your API versioning is wrong, which is why I decided to do it 3 different wrong ways. *troyhunt.com*, February 2014. Archived at perma.cc/9DSW-DGR5

[43] Brandur Leach. APIs as infrastructure: future-proofing Stripe with versioning. *stripe.com*, August 2017. Archived at perma.cc/L63K-USFW

[44] Alexandre Alves, Assaf Arkin, Sid Askary, et al. Web Services Business Process Execution Language Version 2.0. *docs.oasis-open.org*, April 2007.

[45] What is a Temporal Service? *docs.temporal.io*, 2024. Archived at perma.cc/32P3-CJ9V

[46] Stephan Ewen. Why we built Restate. *restate.dev*, August 2023. Archived at perma.cc/BJJ2-X75K

[47] Keith Tenzer and Joshua Smith. Idempotency and Durable Execution. *temporal.io*, February 2024. Archived at perma.cc/9LGW-PCLU

[48] What is a Temporal Workflow? *docs.temporal.io*, 2024. Archived at perma.cc/B5C5-Y396

[49] Jack Kleeman. Solving durable execution's immutability problem. *restate.dev*, February 2024. Archived at perma.cc/G55L-EYH5

[50] Srinath Perera. Exploring Event-Driven Architecture: A Beginner's Guide for Cloud Native Developers. *wso2.com*, August 2023. Archived at archive.org

[51] Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed Virtual Actors for Programmability and Scalability. Microsoft Research Technical Report MSR-TR-2014-41, March 2014. Archived at perma.cc/PD3U-WDMF