Chapter 4. Encoding and Evolution

- Applications inevitably change over time. Features are added or modified (new field or record type needs to be captured, data needs to be presented in a new way, etc.).
- Different data models have different ways of coping with such change.
- Relational databases generally assume that all data in the database conforms to one schema
- When a data format or schema changes, a corresponding change to application code often needs to happen.
- Server-side applications can perform a rolling upgrade (also known as a staged rollout), deploying the new version to a few nodes at a time.
- Client-side applications changes are done by 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. This requires:
    - **Backward compatibility -** newer code can read data that was written by older code.
    - **Forward compatibility - o**lder code can read data that was written by newer code.

Formats for Encoding Data

- Programs usually work with data in (at least) two different representations:
    - Memory - data is kept in objects, structs, lists, arrays, hash tables, trees, etc.
    - Disk - data to a file (or send it over the network) is encoded as self-contained sequence of bytes.
- This requires translation:
    - **Encoding** - translation from the in-memory representation to a byte sequence (also known as serialization* or marshalling)
    - **Decoding** - - translation from the to a byte sequence representation to an in-memory representation (also known as parsing, deserialization, unmarshalling)
- Terminology clash*

Language-Specific Formats

- Many programming languages come with built-in support for quick and easy encoding in-memory objects into byte sequences. (Java - java.io.Serializable, Ruby - Marshal, Python – pickle)
- Allow in-memory objects to be saved and restored with minimal additional code.

Problems:

- The encoding is often tied to a particular programming language and reading the data in another language is very difficult. This requires a long-time commitment.
- The decoding process needs to be able to instantiate arbitrary classes. This is frequently a source of security problems
- Versioning data is often an afterthought in these libraries (neglect the inconvenient problems of forward and backward compatibility).
- Efficiency (CPU time taken to encode or decode, and the size of the encoded structure) is also often an afterthought

Conclusion – don't use a language's built-in encoding for anything other than transient purposes.

JSON, XML, and Binary Variants

- Standardized encodings that can be written and read by many programming languages (i.e. JSON and XML documents)
- JSON, XML, and CSV are textual formats which contain ambiguity around the encoding of numbers.
- XML and CSV cannot distinguish between a number and a string that happens to consist of digits
- JSON distinguishes strings and numbers, but it doesn't distinguish integers and floating-point numbers, and it doesn't specify a precision.
- 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).
- XML and JSON have schema support – ensure quality of client submitted data
- CSV does not have any schema - application must define the meaning of each row and column

Binary encoding

- When working with large amounts of data, the choice of data format can have a big impact.

- JSON and XML use a lot of space compared to binary formats

Example 4-1. Example record which we will encode in several binary formats in this chapter

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

MessagePack

Byte sequence (66 bytes):

```
83 a8 75 73 65 72 4e 61 6d 65 a6 4d 61 72 74 69 6e ae 66 61
76 6f 72 69 74 65 4e 75 6d 62 65 72 cd 05 39 a9 69 6e 74 65
72 65 73 74 73 92 ab 64 61 79 64 72 65 61 6d 69 6e 67 a7 68
61 63 6b 69 6e 67
```

Breakdown:

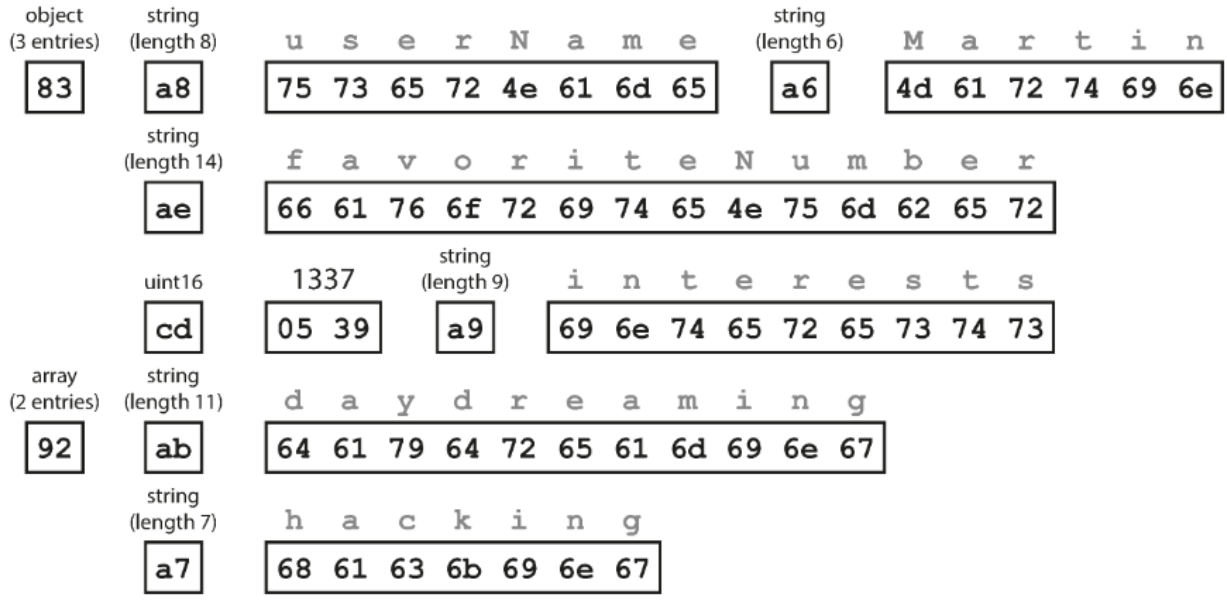

Figure 4-1. Example record (Example 4-1) encoded using MessagePack.

- This binary encoding is 66 bytes long, which is only a little less than the 81 bytes taken by the textual JSON encoding

Thrift and Protocol Buffers

- Apache Thrift and Protocol Buffers 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.
- Both Thrift and Protocol Buffers require a schema for any data that is encoded.

  Thrift schema example:

  ```
  struct Person {
          1: required string      userName,
          2: optional i64         favoriteNumber,
          3: optional list<string> interests
  }
  ```

  Protocol Buffers example:

  ```
  message Person {
          required string user_name       = 1;
  ```

```
        optional int64  favorite_number = 2;
        repeated string interests       = 3;
    }
```

- Thrift and Protocol Buffers each come with a code generation tool.
- Thrift has two different binary encoding formats called BinaryProtocol and CompactProtocol,.

Binary Protocol

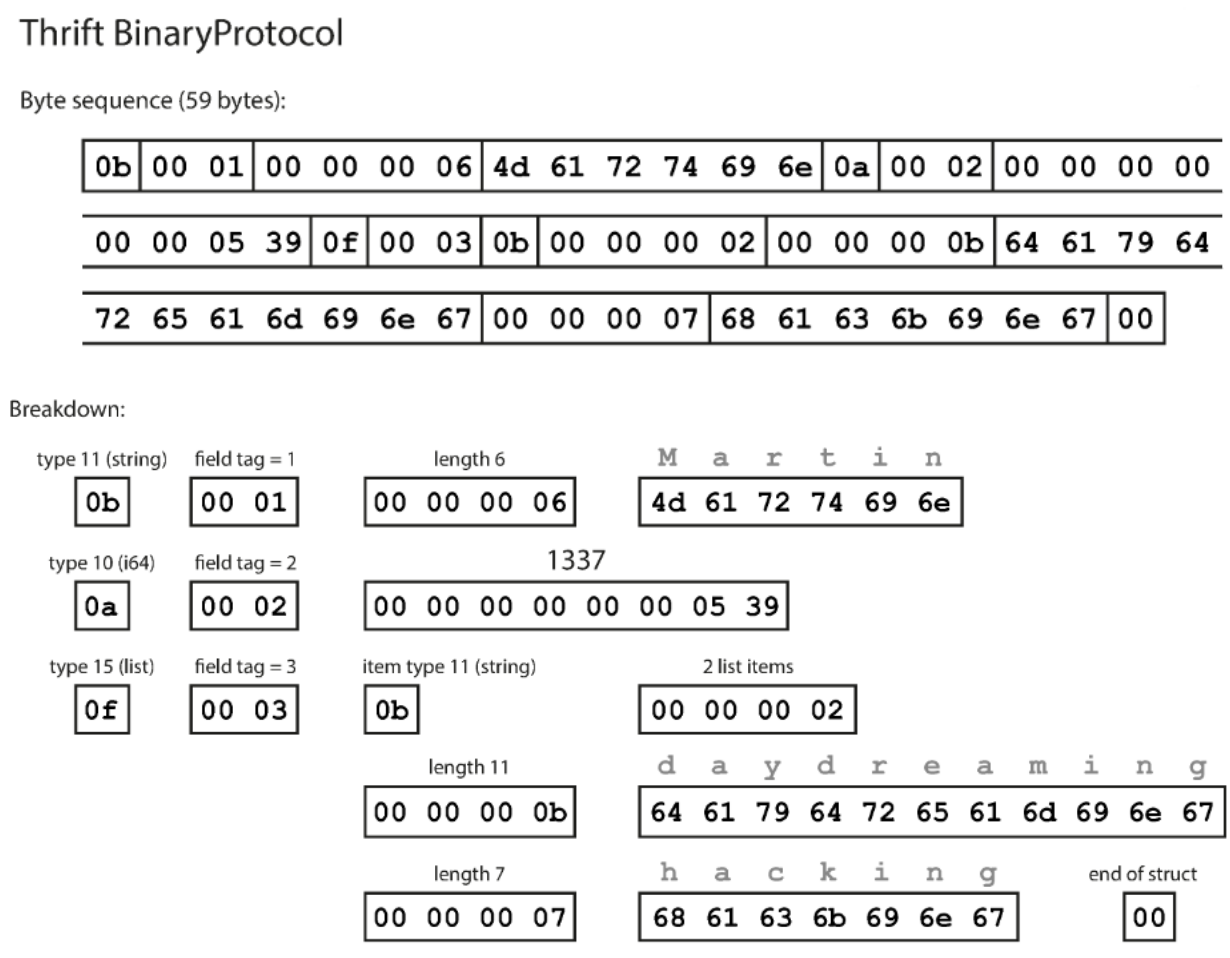

Figure 4-2. Example record encoded using Thrift's BinaryProtocol.

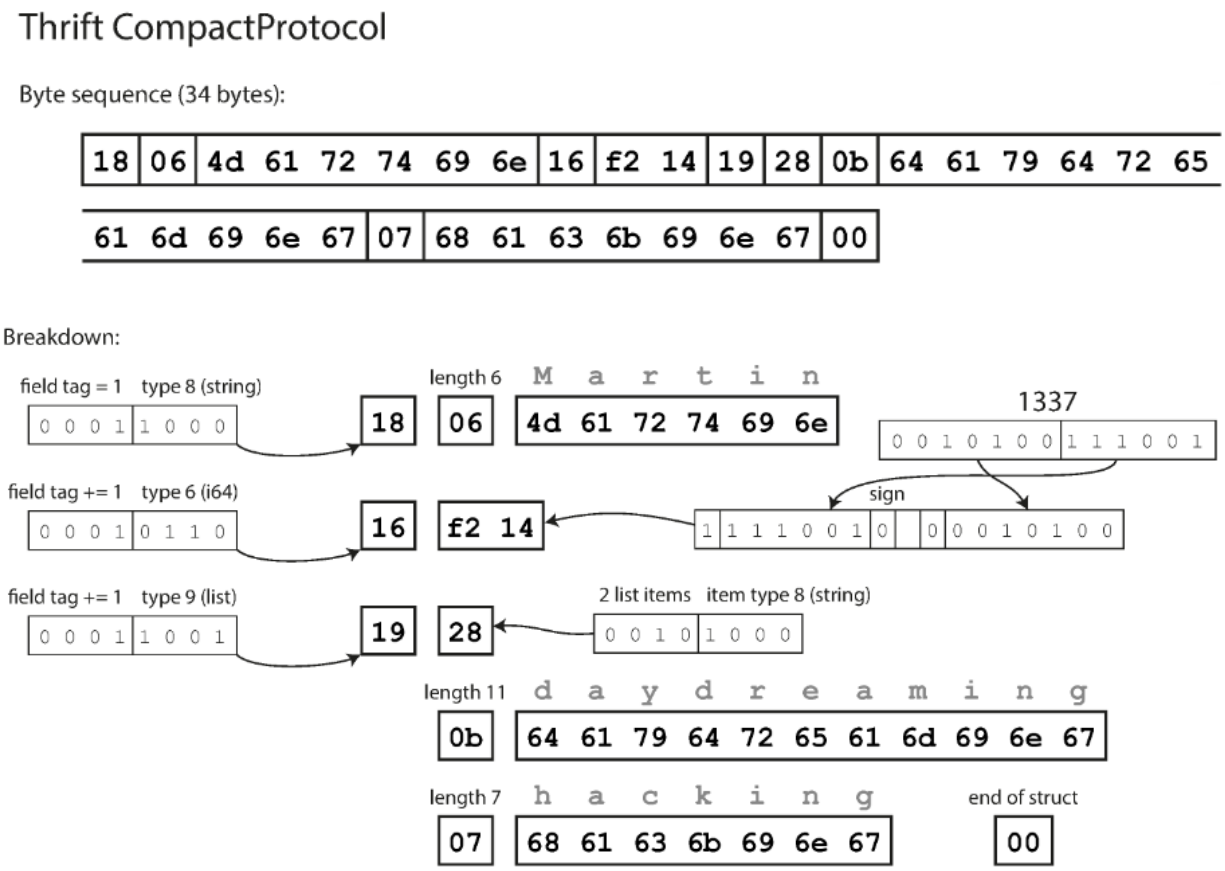

Figure 4-3. Example record encoded using Thrift's CompactProtocol.

- Schemas shown each contain field that was marked either required or optional, but this makes no difference to how the field is encoded (nothing in the binary data indicates whether a field was required). The difference is simply that required enables a runtime check that fails if the field is not set, which can be useful for catching bugs.

Field tags and schema evolution

- Schema evolution - schemas need to change over time.
- Field name in schema can change you cannot change a field's tag, since that would make all existing encoded data invalid.
- Forward compatibility - new fields can be added to the schema if you give each field a new tag number. Old code can simply ignore that field.
- Backward compatibility - If each field has a unique tag number, new code can always read old data, because the tag numbers still have the same meaning. However, new field, you cannot make it required.

- Removing a field – works just like adding a field, with backward and forward compatibility concerns reversed (can only remove a field that is optional and can never use the same tag number again

Datatypes and schema evolution

- Datatype of a field may change (i.e.. 32-bit integer to 64-bit)
- Protocol buffer does not have a list or array datatype, but instead has a repeated marker for fields
- Thrift has a dedicated list datatype, which is parameterized with the datatype of the list elements.

Avro

- Apache Avro 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.
- 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.

```
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", "long"], "default": null},
      {"name": "interests",     "type": {"type": "array", "items": "string"}}
    ]
}
```

Avro

Byte sequence (32 bytes):

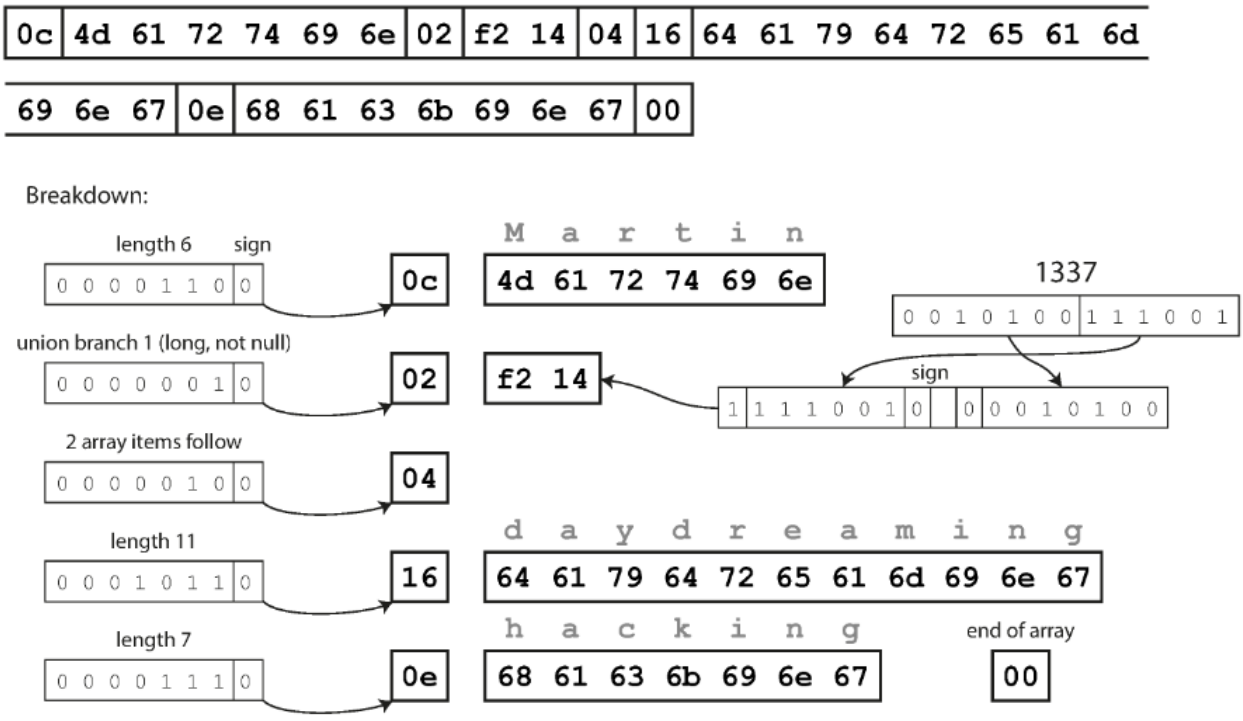

Figure 4-5. Example record encoded using Avro.

- Doesn't use tag numbers in the schema.
- Doesn't associate fields with their datatypes.
- Encoding simply consists of values concatenated together. (i.e. 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)
- 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.
- Support schema evolution by using writer's schema and the reader's schema
    - writer's schema - encodes the data using whatever version of the schema it knows about
    - reader's schema – format expected by an application wants to decode some data
    - Avro does not require that the writer's schema and the reader's schema be the same—they only need to be compatible.
    - An Avro reader resolves differences between the writer's schema and the reader's schema.

Schema evolution rules

- forward compatibility - you can have a new version of the schema as writer and an old version of the schema as reader.

- backward compatibility - 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.
- Changing the datatype of a field is possible, if Avro can convert the type.
- Changing the name of a field is possible but a little tricky
- Unanswered question: how does the reader know the writer's schema with which a particular piece of data was encoded?
    - Large file with lots of records - large file containing millions of records, all encoded with the same schema have the writer of that file 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 - 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.
    - Sending records over a network connection – communicating processes negotiate the schema version on connection setup and then use that schema for the lifetime of the connection. The Avro RPC protocol works like this.
- Compared to Protocol Buffers and Thrift, Avro is friendlier to dynamically generated because doesn't contain any tag numbers.
- Thrift and Protocol Buffers rely on code generation. Because Avro can dynamically generate a schema, code generation is an unnecessary obstacle.
- Avro does provide optional code generation for statically typed programming languages, but it can be used just as well without any code generation.

The Merits of Schemas

- Protocol Buffers, Thrift, and Avro all use a simple schema to describe a binary encoding format.
- XML Schema or JSON Schema which support much more detailed validation rules
- Although textual data formats such as JSON, XML, and CSV are widespread, binary encodings based on schemas are also a viable option. They have several 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.

Modes of Dataflow

- 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 asynchronous message passing (see “Message-Passing Dataflow”)

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 or different processes accessing the database at the same time.
- Could result in different versions of the code reading or writing simultaneously so forward and backward compatibility is necessary.

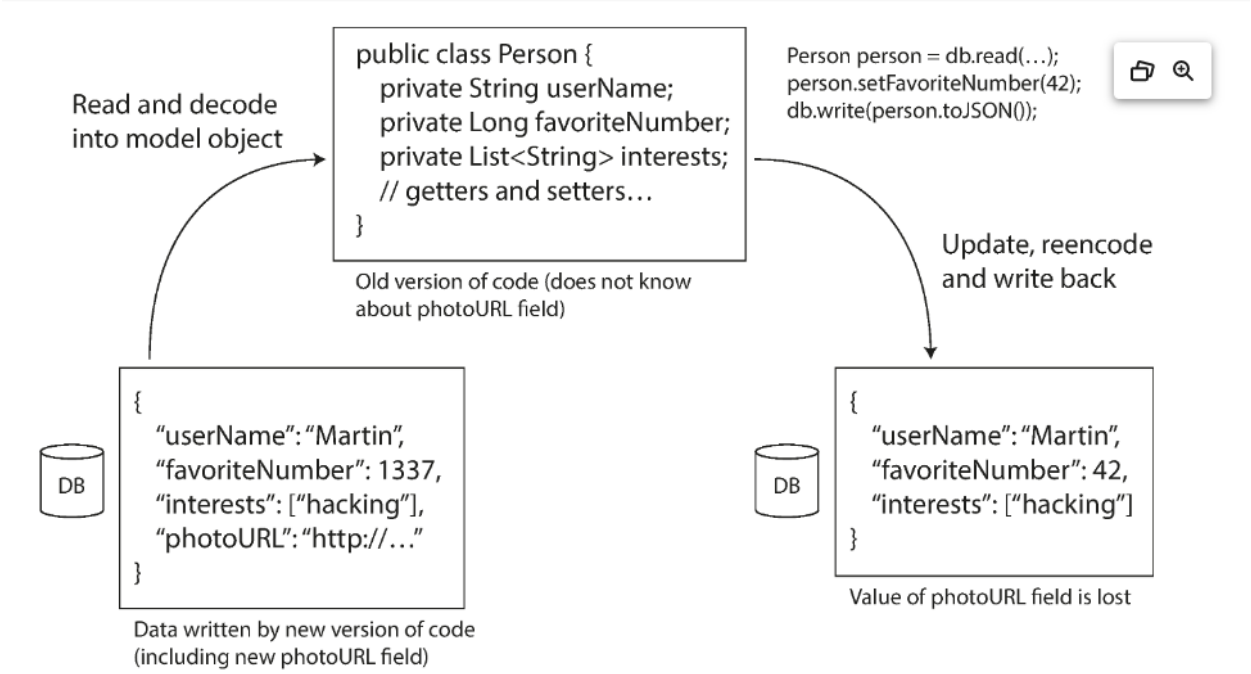

Figure 4-7. When an older version of the application updates data previously written by a newer version of the application, data may be lost if you’re not careful.

Different values written at different times

- Data outlives code.
- Rewriting (migrating) data into a new schema is possible, but it’s an expensive thing to do on a large dataset.
- Most relational databases allow simple schema changes, such as adding a new column with a null default value, without rewriting existing data.
- Schema evolution 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.

Archival storage

- Snapshot of database (data dump) from time to time for backup purposes or for loading into a data warehouse
- Data dump will typically be encoded using the latest schema
- Data dump is immutable

Dataflow Through Services

- Processes that need to communicate over a network are considered clients and servers.
- The servers expose an API over the network. This exposed API
- 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 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.

Web services

- When HTTP is used as the underlying protocol for talking to the service, it is called a web service.
- There are two popular approaches to web services: REST and SOAP.
- REST is not a protocol, but rather a design philosophy that builds upon the principles of HTTP. It emphasizes simple data formats, using URLs for identifying resources and using HTTP features for cache control, authentication, and content type negotiation.
- SOAP is an XML-based protocol (Web Services Description Language (WSDL)) for making network API requests. Although commonly used over HTTP, it aims to be independent from HTTP and avoids using most HTTP features.

The problems with remote procedure calls (RPCs)

- The RPC model has been around since the 1970s. It tries to make a request to a remote network service look the same as calling a function or method in your programming language
- Although RPC seems convenient at first, the approach is fundamentally flawed as 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, in a predictable amount of time.
- 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.

Current directions for RPC

- Despite all these problems, RPC isn't going away. Various RPC frameworks have been built on top of all the encodings previously mentioned
- New generation of RPC frameworks:

    - More explicit about the fact that a remote request is different from a local function call.
    - gRPC supports streams, where a call consists of not just one request and one response, but a series of requests and responses over time
    - Utilizes service discovery which allows a client to find out at which IP address and port number it can find a particular service.
    - Custom RPC protocols with a binary encoding format can achieve better performance than something generic like JSON over REST.
- Generic APIs still have significant advantages
    - good for experimentation and debugging
    - supported by all mainstream programming languages and platforms
    - vast ecosystem of tools available (servers, caches, load balancers, proxies, firewalls, monitoring, debugging tools, testing tools, etc.).

Data encoding and evolution for RPC

- For evolvability, it is important that RPC clients and servers can be changed and deployed independently.
- The backward and forward compatibility properties of an RPC scheme are inherited from whatever encoding it uses.
- Service compatibility is made harder by the fact that RPC is often used for communication across organizational boundaries, so compatibility needs to be maintained for a long time, perhaps indefinitely.

Message-Passing Dataflow

- Asynchronous message-passing systems - somewhere between RPC and databases.
- A client's request (usually called a message) is delivered to another process with low latency.
- They are like 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).
- Big difference compared to RPC is that message-passing communication is usually one-way

Message brokers

- When a process sends a message to a named queue or topic, and the broker ensures that the message is delivered to one or more consumers of or subscribers to that queue or topic. There can be many producers and many consumers on the same topic.
- A topic provides only one-way dataflow.
- Message brokers typically don't enforce any particular data mode

Distributed actor frameworks

- The actor model is a programming model for concurrency in a single process.
- Rather than dealing directly with threads 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 distributed actor frameworks, this programming model is used to scale an application across multiple nodes.
- 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.
- Distributed actor framework essentially integrates a message broker and the actor programming model into a single framework.
- When perform rolling upgrades on actor-based applications, forward and backward compatibility is an issue (message sent or received from nodes running old applications)
- Frameworks that handle message encoding:
    - Akka - uses Java's built-in serialization by default, which does not provide forward or backward compatibility. Can use Protocol Buffers to gain the ability to do rolling upgrades.
    - Orleans - supports rolling upgrades using its own versioning mechanism. It allows new actor methods to be defined (that is, new types of incoming message that an actor can process) while maintaining backward compatibility, provided that existing methods are not changed.
    - Erlang OTP - surprisingly hard to make changes to record schemas (despite the system having many features designed for high availability); rolling upgrades are possible but need to be planned carefully. An experimental new maps datatype (a JSON-like structure, introduced in Erlang R17 in 2014) may make this easier in the future.