Distributed Systems Lab 07

Data serialization & deserialization

During the previous lab, we discussed stable storage and presented how to implement it on top of a filesystem. This lab discusses data serialization and deserialization, which are commonly used when different components of a distributed system require exchanging complex data structures. We will use files contained in this package, which you should download and extract locally.

Moreover, this lab features no Small Assignment. Instead, you will be presented the second Large Assignment.

Learning Section

Data serialization and deserialization

When an in-memory object has to be transmitted (e.g., over a network to another component of a distributed system) or stored somewhere (e.g., in a file), it has to be transformed into a suitable data format.

A simple approach would be to simply use the in-memory representation. While Rust data layout is not stable (even across builds), we could enforce the lingua-franca layout of C with `#repr(C)` attribute:

```rust
#[repr(C)]
struct CStruct {
    a: u32,
    b: u8,
    c: [u16; 3],
}
```

However, this format is useful only as long as the type does not contain any pointers or references. Moreover, transmutation of bytes into a structure is considered an unsafe operation in Rust.

In general, such a process is called **serialization** (or **marshaling**, although in some contexts the terms might not be considered to be exact synonyms). The reverse operation is named **deserialization**.

There are multiple data formats used for serialization. They can be roughly divided along two axes:

- **text-based** vs **binary** formats,
- having a required vs optional **schema** to make sense of the encoding.

For example, JSON is a well-known *text-based* format, very popular especially in the context of fronted development or within APIs serving requests external to the system. Data encoded as JSON is easily understandable by humans, which greatly facilitates debugging. However, for internal communication of specialized software, especially when large volumes of data are processed, *binary* formats are usually preferable, as they offer a more compact encoding. Some commonly known

binary formats are: Protocol Buffers (a.k.a. protobuf), Apache Avro, and MessagePack (compact JSON).

From the *schema* viewpoint, since both JSON and MessagePack always represent data as a combination of primitive types: numbers, strings, lists, mappings, etc., the receiver is not required to know the exact type of the message to interpret the bytes. (Although, multiple tools exist to define what constitutes a valid message.) The downside of *schema-less* formats is explicit typing of all constituents of complex data, which could be avoided if sides of communication shared a specification of the bytes stream. For instance, the `C` representation of the aforementioned structure is such a schema. Similarly, to use Protocol Buffers in a distributed system, we need to share the messages specification (`.proto` files) between components.

During this course, we will use the bincode crate for serialization between Rust programs. The crate implements a custom binary format tailored to Rust (similar to `repr(C)`). But, how do we specify the schema?

Serde

Serde is a framework that facilitates **ser**ialization and **de**serialization of Rust data structures.

Any data structure that implements traits `Serialize` and `Deserialize` can be easily serialized into and deserialized from multiple data formats (see a list on the Serde website), including bincode. To make a custom type serializable, the easiest way is to allow the implementation to be automatically generated (*derived*) by the framework. In our previous example, it is enough to replace the outer attribute:

```rust
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize)]
struct CStruct {
    a: u32,
    b: u8,
    c: [u16; 3],
}
```

An example how to serialize and deserialize data in Rust is provided in `examples/bincode_serde.rs`.

Under the hood

Internally, Serde decouples the type schema (e.g., `CStruct`) from the wire data format (e.g., JSON, bincode) by employing a visitor pattern over intermediate representation (Serde data model).

For serialization, the data format implements the `Serializer` trait handling 29 possible types: primitives, strings, arrays, structures, tuples, maps, unit, etc. The data structure implements the `Serialize` trait with a single method: `fn serialize<S: Serializer>(&self, serializer: S) → Result<S::Ok, S::Error>`, which encodes the `Self` type into the Serde data model by a series of calls to `Serializer` methods corresponding to the intermediate types.

Conversely, for deserialization, the data structure implements `Deserialize` and `Visitors` with methods for handling all possible intermediate types. The data format implements `Deserializer`, accepting a byte sequence, which support two ways of operation:

- as reflected to serialization (for self-describing types), where the `Deserializer` drives the operation,

- as analogous to serialization, where the `Deserialize` trait implementation drives the procedure.

Additional Homework

Think about the safety of the deserialization procedure. Why **transmutation** (casting) from `&[u8]` into `#repr(C) T` is unsafe even for relatively simple types? How can Serde guarantee a safe abstraction? For instance, you may consider an enumeration with an invalid tag number.

Moreover, you can learn about Serde limitations: what data formats cannot be represented?

Authors: F. Plata, K. Iwanicki, M. Banaszek, W. Ciszewski, M. Matraszek