CUSTOM SERIALIZATION WITH SERDE

Supporting the ROS2 message format in Dora

Philipp Oppermann
os2edu
2023-11-24

Agenda

- Introduction to `serde` crate
  - Simple serialization using `derive` attributes
  - The `Serialize` and `Deserialize` traits in detail
  - Example implementations
- Reading and writing ROS2 messages in Dora
  - Serializer with dynamic target type
  - Dynamic deserialization using `DeserializeSeed`

The `serde` crate

- Framework for serializing and deserializing Rust data structures
- Based on Rust's trait system
- Independent of data format
  - Generic data format that can represent all Rust types
  - External crates to translate `serde` data format into target format
    - E.g. for JSON, YAML, MessagePack, TOML, CSV, binary formats, etc.
- Derive macros for convenience
- Supports `no_std` environments

Serde Derive Example

```
#[derive(serde::Serialize, serde::Deserialize, Debug, PartialEq, Eq)]
enum Shape {
    Point(Point),
    Line { from: Point, to: Point },
}

#[derive(serde::Serialize, serde::Deserialize, Debug, PartialEq, Eq)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let shape = Shape::Line {
        from: Point { x: 1, y: 2 },
        to: Point { x: 3, y: 4 },
    };

    let serialized = serde_json::to_string_pretty(&shape).unwrap();
    println!("{serialized}");
    assert_eq!(serde_json::from_str::<Shape>(&serialized).unwrap(), shape);
}
```

Output:

```
{
  "Line": {
    "from": {
      "x": 1,
      "y": 2
    },
    "to": {
      "x": 3,
      "y": 4
    }
  }
}
```

Serde Derive Example: YAML

```rust
#[derive(serde::Serialize, serde::Deserialize, Debug, PartialEq, Eq)]
enum Shape {
    Point(Point),
    Line { from: Point, to: Point },
}

#[derive(serde::Serialize, serde::Deserialize, Debug, PartialEq, Eq)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let shape = Shape::Line {
        from: Point { x: 1, y: 2 },
        to: Point { x: 3, y: 4 },
    };

    let serialized = serde_yaml::to_string(&shape).unwrap();
    println!("{serialized}");
    assert_eq!(serde_yaml::from_str::<Shape>(&serialized).unwrap(), shape);
}
```

YAML Output:

```yaml
---
Line:
  from:
    x: 1
    y: 2
  to:
    x: 3
    y: 4
```

Serde Derive: Attributes

Derived `Serialize`/`Deserialize` implementations can be customized through `#[serde]` attributes, e.g.:

- Rename containers or fields:

```rust
#[derive(serde::Serialize, serde::Deserialize)]
#[serde(rename = "GreatExample", rename_all = "camelCase")]
struct Example {                           // struct name will be changed to GreatExample
    field_one: u32,                        // will be renamed to fieldOne
    some_other_field: u32,                 // will be renamed to someOtherField
    #[serde(rename = "lastField")]
    field_three: bool,                     // will be renamed to lastField
}
```

Serde Derive: Attributes

Derived Serialize/Deserialize implementations can be customized through #[serde] attributes, e.g.:

- Rename containers or fields:

```rust
#[derive(serde::Serialize, serde::Deserialize)]
#[serde(rename = "GreatExample", rename_all = "camelCase")]
struct Example {                          // struct name will be changed to GreatExample
    field_one: u32,                       // will be renamed to fieldOne
    some_other_field: u32,                // will be renamed to someOtherField
    #[serde(rename = "lastField")]
    field_three: bool,                    // will be renamed to lastField
}
```

- Don't serialize None fields:

```rust
#[derive(serde::Serialize, serde::Deserialize)]
struct Example {
    #[serde(skip_serializing_if = "Option::is_none")]
    field: Option<u32>,
}
```

Serde Derive: More Attributes

- Flatten nested structs:

```rust
#[derive(serde::Serialize, serde::Deserialize)]
struct Example {
    field_1: u32,
    #[serde(flatten)]
    field_2: Field2,
}

#[derive(serde::Serialize, serde::Deserialize)]
struct Field2 {
    field_a: bool,
    field_b: String,
}
```

With `flatten`:

```json
{
    "field_1": 42,
    "field_a": true,
    "field_b": "Hello"
}
```

Without `flatten`:

```json
{
    "field_1": 42,
    "field_2": {
        "field_a": true,
        "field_b": "Hello"
    }
  }
```

Serde Derive: More Attributes

- Flatten nested structs:

```
#[derive(serde::Serialize, serde::Deserialize)]
struct Example {
    field_1: u32,
    #[serde(flatten)]
    field_2: Field2,
}

#[derive(serde::Serialize, serde::Deserialize)]
struct Field2 {
    field_a: bool,
    field_b: String,
}
```

- Fill default values on deserialization:

```
#[derive(serde::Serialize, serde::Deserialize)]
struct Example {
    #[serde(default)]
    name: String,                    // fills with empty string if not present
}
```

With `flatten`:

```
{
    "field_1": 42,
    "field_a": true,
    "field_b": "Hello"
}
```

Without `flatten`:

```
{
    "field_1": 42,
    "field_2": {
        "field_a": true,
        "field_b": "Hello"
    }
  }
```

The `Serialize` Trait

```
pub trait Serialize {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
       where S: Serializer;
}
```

- Implemented for Rust types that can be serialized
  - maps Rust types to *serde data model*
- `Serialize` trait is independent of target format
  - generic parameter `S` allows passing in any compatible serializer
- Standard implementations can be derived
  - derive behavior can be customized through attributes
  - full manual implementation is possible too

Serde Data Model

Most Rust types have corresponding types in the serde data model, e.g.:

- all primitive types, e.g. `i64`, floats, `bool`, `char`
- UTF8 strings, byte arrays, `Vec`, `HashMap`
- structs, enums, tuples

→ Serialization is often a trivial mapping

Serde Data Model

Most Rust types have corresponding types in the serde data model, e.g.:

- all primitive types, e.g. `i64`, floats, `bool`, `char`
- UTF8 strings, byte arrays, `Vec`, `HashMap`
- structs, enums, tuples

→ Serialization is often a trivial mapping

`Serialize` implementations can **map to a different serde type** if desired. For example:

- `std::ffi::OsString` represents a platform-native string
    - not guaranteed to be UTF8-encoded, so serializing as serde string would be invalid
    - serialize as ASCII **byte array** on UNIX
    - serialize as **sequence of 16-bit UTF-16** values on Windows

Manual `Serialize` implementation

```rust
struct Point {
    x: i32,
    y: i32,
}

impl serde::Serialize for Point {
    fn serialize<S>(&self, serializer: S,) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        let mut s = serializer.serialize_struct(
            "Point",
            2, // number of fields
        )?;
        s.serialize_field("x", &self.x)?;
        s.serialize_field("y", &self.y)?;
        s.end()
    }
}
```

Example: Change type on `Serialize`

Serialize struct as string:

```rust
struct Point {
    x: i32,
    y: i32,
}

impl serde::Serialize for Point {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        let string_representation = format!("{}:{}", self.x, self.y);
        serializer.serialize_str(&string_representation)
    }
}
```

The `Deserialize` Trait

```
pub trait Deserialize<'de>: Sized {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
       where D: Deserializer<'de>;
}
```

- Goal: Drive the given `Deserializer` to map raw data into the serde data model
- `Deserializer` provides two kinds of entry points:
  - **`deserialize_any`** for self-describing formats (e.g. JSON)
    - instructs the deserializer to read the data type from the raw data
    - panics for formats that are not self-describing (e.g. Cap'n Proto)
  - **typed `deserialize_*` functions**, e.g. `deserialize_f32` or `deserialize_string`

The `Deserialize` Trait

```
pub trait Deserialize<'de>: Sized {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
       where D: Deserializer<'de>;
}
```

- Goal: Drive the given `Deserializer` to map raw data into the serde data model
- `Deserializer` provides two kinds of entry points:
  - **`deserialize_any`** for self-describing formats (e.g. JSON)
    - instructs the deserializer to read the data type from the raw data
    - panics for formats that are not self-describing (e.g. Cap'n Proto)
  - **typed `deserialize_*` functions**, e.g. `deserialize_f32` or `deserialize_string`
- Deserialize methods take a **`Visitor`** implementation that constructs the target value
  - translates the serde data format into a Rust type
  - gives additional type hints for nested data, e.g. structs

Manual `Deserialize` Implementation

```rust
struct Number(u32);

impl<'de> serde::Deserialize<'de> for Number {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
    where
        D: serde::Deserializer<'de>,
    {
        let visitor = NumberVisitor;
        deserializer.deserialize_u32(visitor)
    }
}

struct NumberVisitor;

impl<'de> Visitor<'de> for NumberVisitor {...}          // explained later
```

- we tell the deserializer that we expected a u32 value
    - allows deserializing from formats that are not self-describing

Example: `deserialize_any`

```rust
enum NumberOrString {
    Number(u32),
    String(String),
}

impl<'de> serde::Deserialize<'de> for NumberOrString {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
    where
        D: serde::Deserializer<'de>,
    {
        let visitor = NumberOrStringVisitor;
        deserializer.deserialize_any(visitor)
    }
}

struct NumberOrStringVisitor;
impl<'de> Visitor<'de> for NumberOrStringVisitor {...}         // explained later
```

- only works with self-describing formats, e.g. JSON
- panics when used with other formats, e.g. Cap’n Proto

The `Visitor` trait

```rust
pub trait Visitor<'de>: Sized {
    type Value;     // the target type that we want to deserialize to
    fn expecting(&self, formatter: &mut Formatter<'_>) → Result;       // only required method

    fn visit_bool<E>(self, v: bool) → Result<Self::Value, E> where E: serde::de::Error { ... }
    fn visit_i8<E>(self, v: i8) → Result<Self::Value, E> where E: serde::de::Error { ... }
    fn visit_i16<E>(self, v: i16) → Result<Self::Value, E> where E: serde::de::Error { ... }
    ...
    fn visit_f64<E>(self, v: f64) → Result<Self::Value, E> where E: serde::de::Error { ... }
    fn visit_char<E>(self, v: char) → Result<Self::Value, E> where E: serde::de::Error { ... }
    fn visit_str<E>(self, v: &str) → Result<Self::Value, E> where E: serde::de::Error { ... }
    fn visit_borrowed_str<E>(self, v: &'de str) → Result<Self::Value, E> where E: serde::de::Error { ... }
    fn visit_string<E>(self, v: String) → Result<Self::Value, E> where E: serde::de::Error { ... }
    ...
    fn visit_seq<A>(self, seq: A) → Result<Self::Value, A::Error>
        where A: SeqAccess<'de> { ... }
    fn visit_map<A>(self, map: A) → Result<Self::Value, A::Error>
        where A: MapAccess<'de> { ... }
    fn visit_enum<A>(self, data: A) → Result<Self::Value, A::Error>
        where A: EnumAccess<'de> { ... }
}
```

Example: Creating a `NumberVisitor`

```rust
struct NumberVisitor;

impl<'de> serde::de::Visitor<'de> for NumberVisitor {
    type Value = Number;

    fn expecting(&self, formatter: &mut std::fmt::Formatter) -> std::fmt::Result {
        formatter.write_str("an unsigned 32-bit integer")
    }

    fn visit_u32<E>(self, v: u32) -> Result<Self::Value, E>
    where
        E: serde::de::Error,
    {
        Ok(Number(v))
    }
}
```

- all the `visit_*` methods are optional → deserialization error by default
- we expect a `u32`, so we only need to implement `visit_u32`, right? → **wrong!**

Implementing `visit_u32` is not enough

The following code panics:

Try it yourself on the playground

```
let deserialized: Number = serde_json::from_str("42").unwrap();
// → Error("invalid type: integer '42', expected an unsigned 32-bit integer", line: 1, column: 2)
```

Implementing `visit_u32` is not enough

The following code panics:

[Try it yourself on the playground]

```
let deserialized: Number = serde_json::from_str("42").unwrap();
// → Error("invalid type: integer '42', expected an unsigned 32-bit integer", line: 1, column: 2)
```

- the message after "expected" is from our `expected()` implementation
- the *"invalid type"* error message is coming from the default implementation of **`visit_u64`**
  - why is it called instead of `visit_u32`?
  - remember, we called `deserialize_u32` in our `Deserialize` implementation:

```
impl<'de> serde::Deserialize<'de> for Number {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error> where D: serde::Deserializer<'de> {
        let visitor = NumberVisitor;
        deserializer.deserialize_u32(visitor)     // → leads to a visit_u64 call somehow?
    }
}
```

Fixing the `NumberVisitor` Example

- The `deserialize_*` methods are only a type hint
    - the `Deserializer` is allowed to call other `visit_*` methods
- There are no separate `u8`, `u16`, `u32`, and `u64` types in JSON
    - only a general `unsigned` type
    - → the `serde_json` `Deserializer` will always call `visit_u64` for all unsigned integers

Fixing the `NumberVisitor` Example

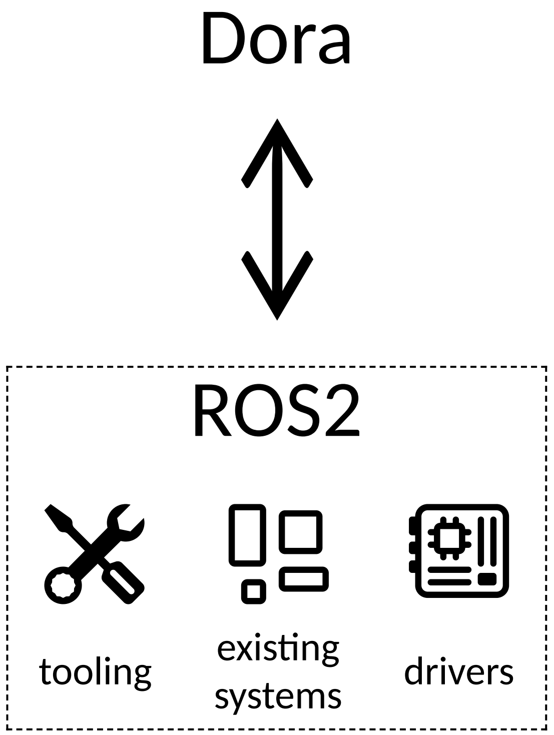
- The `deserialize_*` methods are only a type hint
  - the `Deserializer` is allowed to call other `visit_*` methods
- There are no separate `u8`, `u16`, `u32`, and `u64` types in JSON
  - only a general unsigned type
  - → the `serde_json` `Deserializer` will always call `visit_u64` for all unsigned integers
- **Fix:** Add a `visit_u64` implementation to our `NumberVisitor`:

```rust
fn visit_u64<E>(self, v: u64) -> Result<Self::Value, E> where E: serde::de::Error {
    match u32::try_from(v) {
        Ok(v) => Ok(Number(v)),
        Err(_) => Err(E::custom("number is too large")),
    }
}
```

- Also: implement similar methods, e.g. `visit_u8`, `visit_i32`, and maybe `visit_f32`

Dora ROS2 Bridge

- ROS2 is a mature framework for robot software development
- A bridge to ROS2 has many advantages for Dora
  - Use ROS2 tooling for Dora
    - For example, record and replay Dora messages using `rosbag`
  - Compatibility with existing ROS2 systems
    - Bridge allows partial migration to Dora
  - Drivers written for ROS2
    - Dora nodes can use them through bridge
- Use `ros2-client` crate to
  - communicate with ROS2 nodes via DDS
  - serialize and deserialize messages in CDR format



CDR Format

- short for "Common Data Representation"
- binary format
- not self-describing
  - struct field names etc. are not stored
  - we need to know exact type for deserialization

CDR Format

- short for "Common Data Representation"
- binary format
- not self-describing
  - struct field names etc. are not stored
  - we need to know exact type for deserialization
- ROS2 describes message format in `.msg` files
  - For example, the `Vector3.msg` file looks like this:

    ```
    float64 x
    float64 y
    float64 z
    ```

  - Format is relatively simple → we can parse it
  - Base parser on existing `rclrust-msg-gen` crate

Serializing to dynamic target type

- When using Python nodes, we only know the available ROS2 types at runtime
  - because the `dora-daemon` and `dora-runtime` are precompiled on different machines
  - we still need to serialize them to the **correct ROS2 type**
- Approach:
  - parse ROS2 `.msg` files on initialization
  - supply ROS2 type name on topic creation function → look up type info from `.msg` file
  - on `publish`, use type info to serialize Python object to correct ROS2 type

Serializing to dynamic target type

- When using Python nodes, we only know the available ROS2 types at runtime
  - because the `dora-daemon` and `dora-runtime` are precompiled on different machines
  - we still need to serialize them to the **correct ROS2 type**
- Approach:
  - parse ROS2 `.msg` files on initialization
  - supply ROS2 type name on topic creation function → look up type info from `.msg` file
  - on `publish`, use type info to serialize Python object to correct ROS2 type

Example:

```
turtle_twist_topic = ros2_node.create_topic("/turtle1/cmd_vel", "geometry_msgs::Twist", topic_qos)
twist_writer = ros2_node.create_publisher(turtle_twist_topic)

# serializes 'value' object as 'geometry_msgs::Twist' message
twist_writer.publish(value)
```

Dynamic `Serialize` Implementation

```rust
pub struct TypedValue<'a> {
    pub value: &'a arrow::ArrayData,
    pub type_info: &'a TypeInfo,
}

impl serde::Serialize for TypedValue<'_> {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        match &self.type_info.data_type {
            DataType::UInt8 => {
                let number = convert_to_u8(self.value);     // → this function could e.g. do integer tpe conversion
                serializer.serialize_u8(number)
            }
            DataType::UInt16 => {...}
            ...
        }
    }
}
```

Dynamic `Serialize` Implementation (2)

```
match &self.type_info.data_type {
    // ...continued
    DataType::List(item_info) => {
        let list_array: ListArray = self.value.clone().into();
        let mut s = serializer.serialize_seq(Some(list_array.values.len()))?;

        for item_value in list_array.iter() {
            // leads to recursive call
            let element = TypedValue {
                value: &item_value,
                type_info: &TypeInfo {
                    data_type: item_info.data_type().clone(),
                    // ..omitted
                },
            };
            s.serialize_element(&element)?;
        }

        s.end()
    }
}
```

Dynamic deserialization

- We also want to deserialize ROS2 messages
    - CDR format is not self-describing → cannot use `deserialize_any`
    - we need to use correct `deserialize_*` for each deserialization step
    - → supply type name on subscribe call → get type info of expected type from `.msg` file

Dynamic deserialization

- We also want to deserialize ROS2 messages
    - CDR format is not self-describing → cannot use `deserialize_any`
    - we need to use correct `deserialize_*` for each deserialization step
    - → supply type name on subscribe call → get type info of expected type from `.msg` file
- Challenge: No way to use runtime type info in `Deserialize` implementation:

```
impl<'de> serde::Deserialize<'de> for OwnedTypedValue {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
    where
        D: serde::Deserializer<'de>,
    {
        match type_info.data_type { // -> how to get access to type_info here?
            DataType::UInt32 => deserializer.deserialize_u32(NumberVisitor),
            DataType::List(item_info) => {...}
            ...
        }
    }
}
```

Using `DeserializeSeed`

- It's not possible to use runtime data in `Deserialize` trait
- Fortunately, there is an alternative **`DeserializeSeed`** trait for this case:

```rust
pub trait DeserializeSeed<'de>: Sized {
    type Value;

    fn deserialize<D>(self, deserializer: D) -> Result<Self::Value, D::Error>
        where D: Deserializer<'de>;
}
```

  - runtime context can be stored in `self` object
  - `deserialize` method returns `Self::Value` (instead of returning `Self`)

DeserializeSeed Example

```rust
pub struct TypedDeserializer {
    type_info: TypeInfo,
}

impl<'de> serde::de::DeserializeSeed<'de> for TypedDeserializer {
    type Value = arrow::ArrayData;

    fn deserialize<D>(self, deserializer: D) -> Result<Self::Value, D::Error>
    where
        D: serde::Deserializer<'de>,
    {
        match self.type_info.data_type {
            DataType::UInt32 => deserializer.deserialize_u32(NumberVisitor),
            DataType::List(item_info) => {...}
            ...
        }
    }
}
```

Summary

- The `serde` crate provides format-independent **`Serialize`** and **`Deserialize`** traits
    - Also: **`DeserializeSeed`** for deserialization with additional runtime state
    - Traits map between serde data model and Rust types
    - External crates for mapping serde data model to **different serialization formats**
- **Derive macros** make it easy to add `serde` support for custom structs/enums
    - Attributes allow modifying serialization format, e.g. renames, skipping fields, flattening structs
- Manual `Deserialize` implementations require custom **`Visitor`**
    - **`deserialize_any`** only works with self-describing data formats
    - no guarantee that deserializer follows `deserialize_*` type hint
- Dora supports reading and writing **typed ROS2 messages** using custom serde implementations