An Experimental iRODS Client in Rust

By Phillip Davis

- MsCS, App State ‘24
- Interned with iRODS Summer ‘22 and ‘23
- Currently freelancing

1. Overview
2. Quick tour
3. Housekeeping / Moving forward

Why Rust?

- Memory safety (~= data safety)
- Reliable concurrency.
- Performance.
- Rich type system.
- Absurdly powerful macro system.
- Because I like it!

What's the deal with this client?

- Async everywhere
  - Only runs under Tokio
- Easy to change protocol encoding.
- Designed with longer-running connections in mind.
- Write your rules in Rust.
- (Probably not cross-platform)

```rust
use irods_client::connection::Account;
use irods_client::reexports::tokio;
use irods_client::{connection::authenticate::NativeAu

#[rule(
    body = "writeLine('stdout', '*greeting1, *greetin
    output = "ExecRuleOut"
)]
#[derive(Debug)]
pub struct VeryAdvancedHelloWorldRule {
    pub greeting1: String,
    pub greeting2: String,
}

fn main() {
    let conn = /* Spin up a connection */

    let rule = VeryAdvancedHelloWorldRule {
        greeting1: "Hello".to_string(),
        greeting2: "World".to_string(),
    };

    rule.execute(&mut conn).unwrap();
}
```

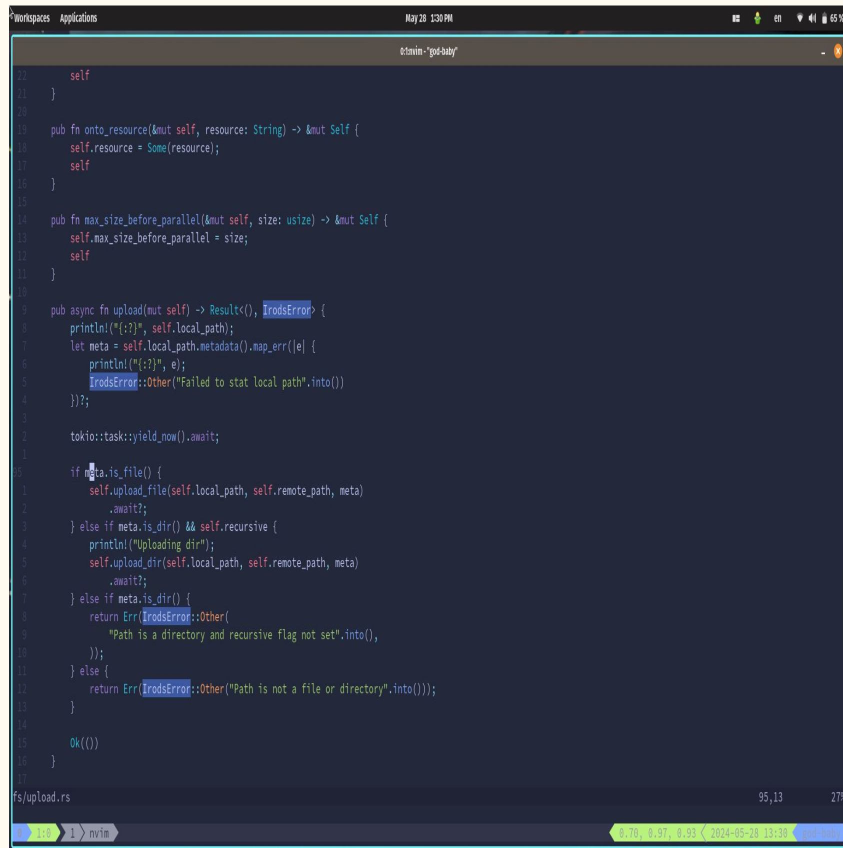
```rust
impl<T, C, A> Manager for IrodsManager<T, C, A>
where
    T: ProtocolEncoding + Send + Sync,
    C: Connect<T> + Send + Sync + 'static,
    C::Transport: Send + Sync + 'static,
    A: Authenticate<T, C::Transport> + Send + Sync + 'static,
{
    type Type = Connection<T, C::Transport>;
    type Error = IrodsError;

    async fn create(&self) -> Result<Self::Type, Self::Error> {
        self.connector
            .connect(self.account.clone())
            .and_then(|unauth_conn| self.authenticator.authenticate(unauth_conn))
            .await
    }

    async fn recycle(
        &self,
        conn: &mut Self::Type,
        metrics: &deadpool::managed::Metrics,
    ) -> RecycleResult<Self::Error> {
        Ok(())
    }
}
```

```rust
    pub(crate) async fn read_standard_header<T>(&mut self) -> Result<StandardHeader, IrodsError>
    where
        T: ProtocolEncoding,
    {
        self.read_to_header_buf(4).await?;
        let header_len = u32::from_be_bytes(self.header_buf[..4].try_into().unwrap()) as usize;
        self.read_to_header_buf(header_len).await?;

        Ok(T::decode(&self.header_buf[..header_len])?)
    }

    pub(crate) async fn read_msg<T, M>(&mut self, len: usize) -> Result<M, IrodsError>
    where
        T: ProtocolEncoding,
        M: Deserializable,
    {
        self.read_to_msg_buf(len).await?;
        Ok(T::decode(&self.msg_buf[..len])?)
    }
```

Caveats / Future Directions

The Bad: or, The Path to Production-Readiness

1. Well-defined error-handling strategy.
2. Comprehensive integration tests.
3. PR in flux with quick-xml.
4. More features (e.g., password-change algorithm)
5. Etc., etc.



The Good: or, Nice-to-haves

1. “Statically” checked rule syntax (via compile-time execution)

2. Statically sized connection buffers might allow arena allocation (inspired by battlesnake_game_types crate)

```rust
mod victor_determinable;
mod you_determinable;

pub use eval::EvaluateMode;

/// A compact board representation that is significantly faster for simulation than
/// `battlesnake_game_types::wire_representation::Game`.
#[derive(Debug, Copy, Clone, PartialEq, Eq, Hash)]
pub struct CellBoard<
    T: CN,
    DimensionsType: Dimensions,
    const BOARD_SIZE: usize,
    const MAX_SNAKES: usize,
> {
    hazard_damage: u8,
    cells: [Cell<T>; BOARD_SIZE],
    healths: [u8; MAX_SNAKES],
    heads: [CellIndex<T>; MAX_SNAKES],
    lengths: [u16; MAX_SNAKES],
    dimensions: DimensionsType,
}

#[allow(dead_code)]
fn get_snake_id(
    snake: &crate::wire_representation::BattleSnake,
```

Join me!

- Recently open-sourced the project. It lives here: https://tinyurl.com/irods-rust-github

- Issue #7 is a (subject-to-change) list of issues that are important to resolve before issuing an initial release.

- Anyone who wants an iRODS Rust client is welcome to contribute. Right now it’s just me.

Thank you.