RosLibRust

Easy way to talk to any ROS system from Rust

Built by: https://github.com/Carter12s & https://github.com/ssnover

Rust in a Nutshell

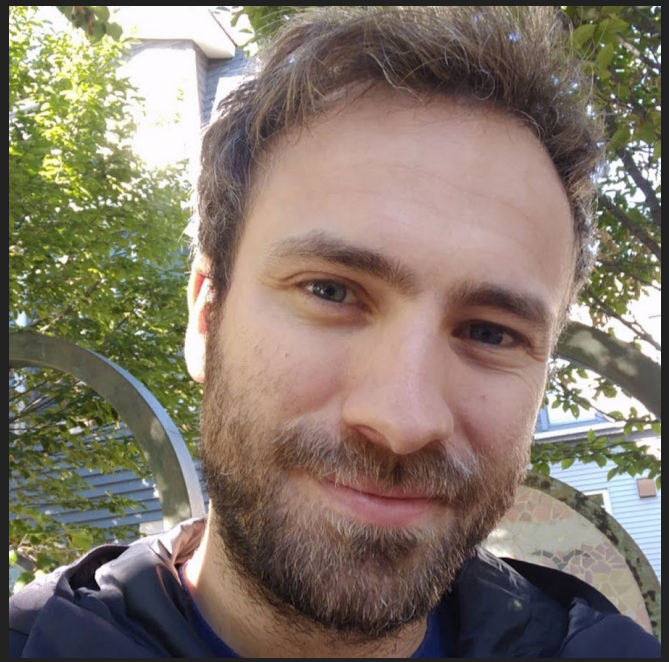
“Fast, Reliable, Productive – Pick Three!”

“Fearless Concurrency”

“Batteries Included”

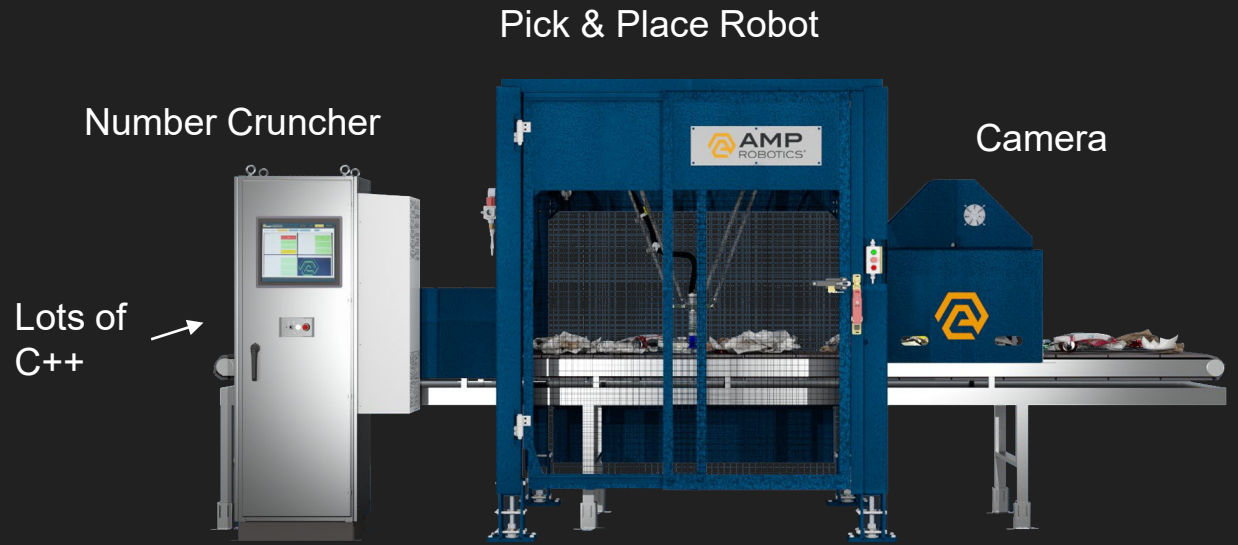
Considering Rust

by John Gjengset

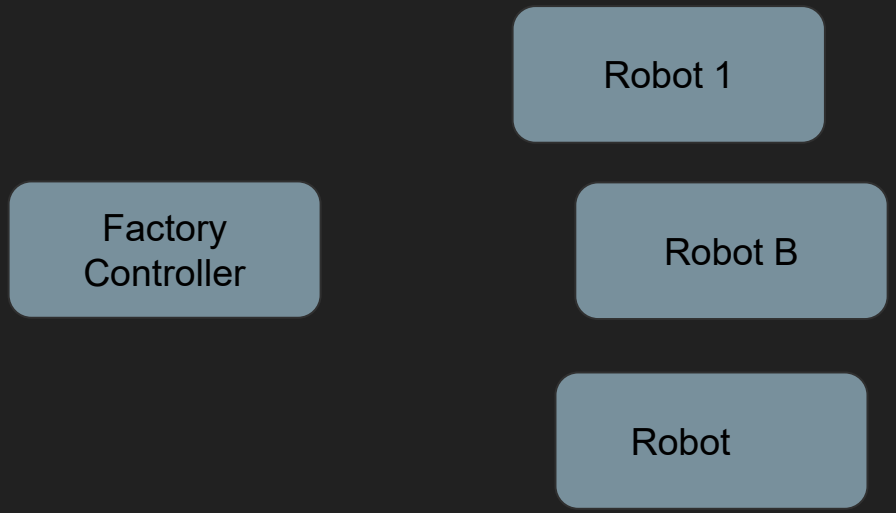


AMP Robotics

- 200+ Systems operating around the world
- Started building our own sorting facilities around this tech



The Problem: Connecting to Dozens of Systems

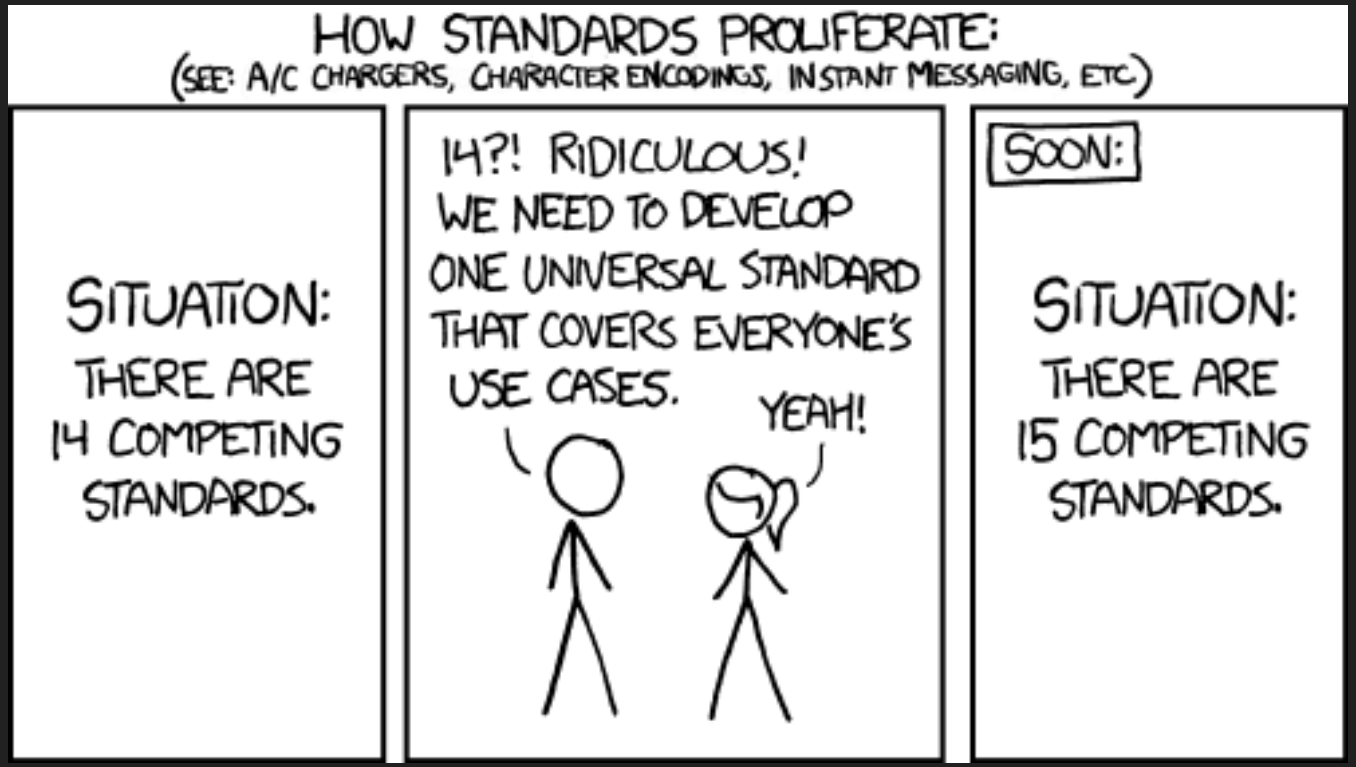


The ROS + Rust Ecosystem is messy…

- ROS1:
  - rosrust ~639

Problems we faced:

1. All the crates have incompatible APIs
2. Some crates require a ROS installation, some are standalone
3. All crates have “strict message parsing”
4. We really like async! No async ROS1 crate exists yet…


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION:
THERE ARE
14 COMPETING
STANDARDS.
14?! RIDICULOUS!
WE NEED TO DEVELOP
ONE UNIVERSAL STANDARD
THAT COVERS EVERYONE'S
USE CASES.
YEAH!
SOON:
SITUATION:
THERE ARE
15 COMPETING
STANDARDS.

Not actually that bad…

Designed to run on ROS system:

rospy

roscpp

rosrust

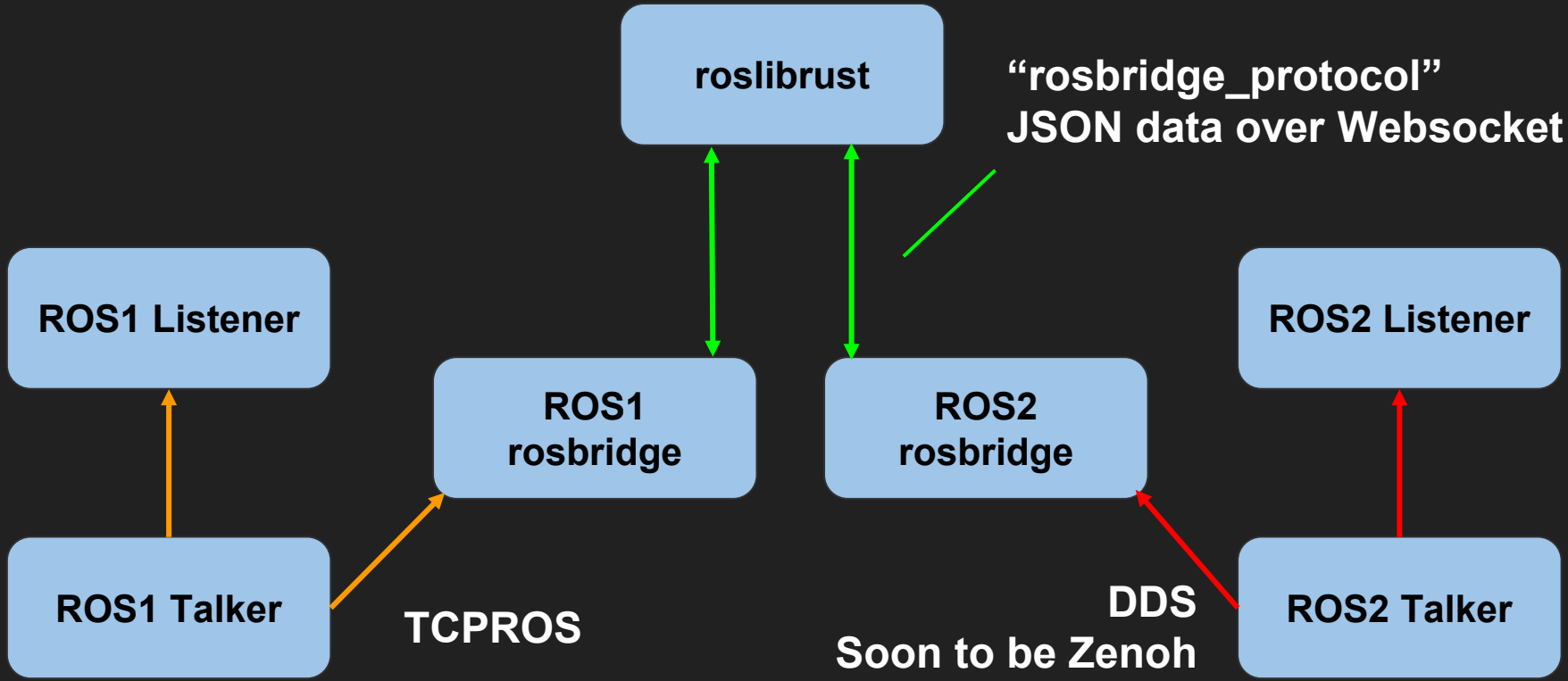
Designed to talk to ROS from non-ROS system:

roslibpy

roslibjs

roslibrust

ROS Bridge!


roslibrust
“rosbridge_protocol”
JSON data over Websocket
ROS1 Listener
ROS1 rosbridge
ROS2 rosbridge
ROS2 Listener
ROS1 Talker
TCPROS
DDS
Soon to be Zenoh
ROS2 Talker

Stop talking and show us the code…

```rust
use roslibrust::ClientHandle;
use roslibrust_codegen_macro::find_and_generate_ros_messages;

find_and_generate_ros_messages!("assets/ros1_common_interfaces/std_msgs");

#[tokio::main(flavor = "multi_thread")]
async fn main() -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    let client = ClientHandle::new("ws://localhost:9090").await?;
    let publisher = client.advertise::<std_msgs::Header>("talker").await?;

    loop {
        let msg = std_msgs::Header::default();
        publisher.publish(msg).await.unwrap();
        tokio::time::sleep(tokio::time::Duration::from_secs(1)).await;
    }
}
```

Procedural macro – Rust code that generates Rust code

```
find_and_generate_ros_messages!("assets/ros1_common_interfaces/std_msgs");
```

- Allows library authors to write “compiler plugins”
- Invoked by compiler during compilation
- Works directly with compiler’s internal source code AST

```
#[proc_macro]
pub fn find_and_generate_ros_messages(input_stream: TokenStream) -> TokenStream {
```

- Recursively searches ROS_PACKAGE_PATH + input
- Find all packages (ROS1 + ROS2)
- Parse all found message, service, action files
- Resolve dependencies
- Generate Rust types

Codegen in action

What the message file looks like:

```
# This message holds the status of an individual component of the robot.
#

# Possible levels of operations
byte OK=0
byte WARN=1
byte ERROR=2
byte STALE=3

byte level # level of operation enumerated above
string name # a description of the test/component reporting
string message # a description of the status
string hardware_id # a hardware unique string
KeyValue[] values # an array of values associated with the status
```

Codegen in action

What gets generated:

Serialization hooks

Data definition

Constants

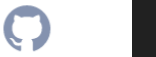
```
#[allow(non_snake_case)]
#[derive(
    :: serde :: Deserialize,
    :: serde :: Serialize,
    :: smart_default :: SmartDefault,
    Debug,
    Clone,
    PartialEq,
)]
pub struct DiagnosticStatus {
    pub r#level: u8,
    pub r#name: ::std::string::String,
    pub r#message: ::std::string::String,
    pub r#hardware_id: ::std::string::String,
    pub r#values: ::std::vec::Vec<self::KeyValue>,
}
impl ::roslibrust_codegen::RosMessageType for DiagnosticStatus {
    const ROS_TYPE_NAME: &'static str = "diagnostic_msgs/DiagnosticStatus";
}
impl DiagnosticStatus {
    pub const r#OK: u8 = 0u8;
    pub const r#WARN: u8 = 1u8;
    pub const r#ERROR: u8 = 2u8;
    pub const r#STALE: u8 = 3u8;
}
```

serde-rs/**serde**

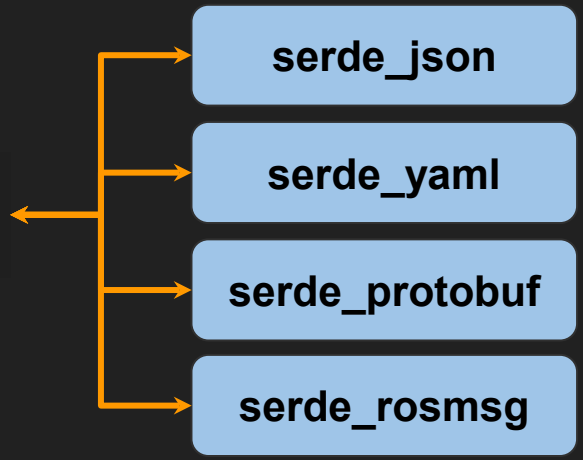
serde

Serialization framework for Rust

169 Contributors | 633k Used by | 8k Stars | 714 Forks



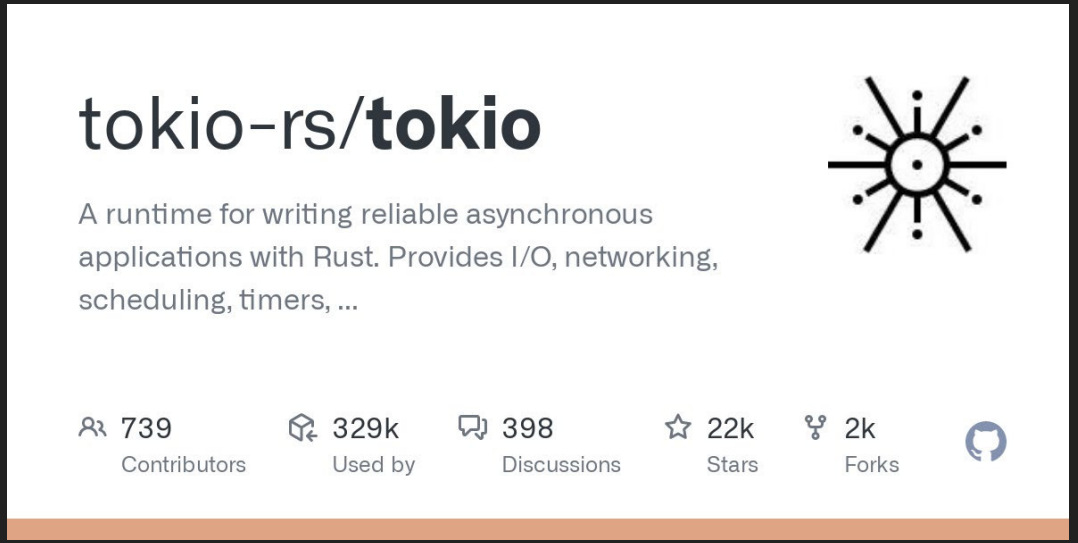
```rust
#[derive(serde::Serialize, serde:Deserialize)]
struct MyStruct {}
```



Kicking off our main function with Async!

```rust
#[tokio::main(flavor = "multi_thread")]
async fn main() -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
```

- Entire application is async
- Runs on a thread pool
- Allows extremely efficient parallelism and concurrency
- Rust compiler prevents all memory safety issues



```rust
let client = ClientHandle::new("ws://localhost:9090").await?;
let publisher = client.advertise::<std_msgs::Header>("talker").await?;

loop {
    let msg = std_msgs::Header::default();
    publisher.publish(msg).await.unwrap();
    tokio::time::sleep(tokio::time::Duration::from_secs(1)).await;
}
```

None of that is new?
Why should we care?

A more interesting example!

```rust
mod ros1 {
    crate::find_and_generate_ros_messages!("assets/ros1_common_interfaces/std_msgs");
}

mod ros2 {
    crate::find_and_generate_ros_messages!("assets/ros2_common_interfaces/std_msgs");
}

#[tokio::main(flavor = "multi_thread")]
async fn main() -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    let ros1_client = ClientHandle::new("ws://localhost:9090").await?;
    let ros2_client = ClientHandle::new("ws://localhost:9091").await?;
    let subscriber = ros1_client
        .subscribe::<ros1::std_msgs::Header>("/bridge_header")
        .await?;
    let publisher = ros2_client.advertise("/bridge_header").await?;
    loop {
        let msg = subscriber.next().await;
        let converted_msg = ros2::std_msgs::Header {
            stamp: msg.stamp,
            frame_id: msg.frame_id,
        };
        publisher.publish(converted_msg).await?;
    }
}
```


A really cool example!


```rust
#[derive(serde::Serialize, serde::Deserialize, Clone, Debug)]
#[serde(untagged)]
enum GenericHeader {
    V1(ros1::std_msgs::Header),
    V2(ros2::std_msgs::Header),
}

impl RosMessageType for GenericHeader {
    const ROS_TYPE_NAME: &'static str = "std_msgs/Header";
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    let client = ClientHandle::new("ws://localhost:9090").await?;
    let rx = client.subscribe::<GenericHeader>("talker").await?;
    loop {
        let msg = rx.next().await;
        match msg {
            GenericHeader::V1(ros1_header) => {
                info!("Got ros1: {ros1_header:?}");
            }
            GenericHeader::V2(ros2_header) => {
                info!("Got ros2: {ros2_header:?}");
            }
        }
    }
}
```

When to use roslibrust?

- You want to connect to many diverse ROS systems
- You only need low bandwidth data exchange
  - Converting back and forth to JSON ain't the fastest thing in the world
- You want to leverage Rust's ecosystem:
  - web servers, databases, security, high performance, badass type system
- Ideal for: fleet monitoring, metrics collection, central facility control, and various cloud tools

Roadmap

- Want to be a part of unifying the Rust experience for ROS! (sorry…)
- Crate is still in beta with an evolving API, but has seen some serious use at AMP and is proven reliable
- Have preliminary support for ROS1 native communication (TCPROS)
  - Will be first ROS1 client to provide an async API
- Sticking with “Rust Idiomatic” for now
  - No dependency on ROS1 / ROS2 installation or on catkin, ament, etc.

Thank you!