Let's Rust in Samba

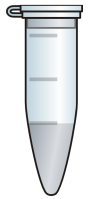
Trying to use Samba with Rust libraries

Kai Blin
Samba Team

SambaXP 2018
2017-06-06

Intro

- M.Sc. in Computational Biology
- Ph.D. in Microbiology
- Samba Team member



Overview

- Rust Intro
- The Example Project
- Challenges
- Conclusions

If someone claims to have the perfect programming language, he is either a fool or a salesman or both.
– Bjarne Stroustrup

Rust Intro

Why?

"The [Samba] project does need to consider the use of other, safer languages."
– Jeremy Allison, SambaXP 2016

Why?

No, honestly, why?

- Avoid whole classes of bugs
- New languages, new features
- It's getting harder to find C programmers

Currently

- 80% C, 7% Python (2)
- ~ 2 million lines of code
- ~ 15-20 regular contributors

Rust

- Announced 2010
- C-like, compiled language
- Focus on memory safety
- Package management with `cargo`
- Still a bit of a moving target
- Programmers call themselves "Rustacians"

Rust

Hello, World!

```
fn main() {
    println!("Hello, world!");
}
```

Introducing the example project.

Plan

- Write a DNS parser in Rust
- Build as a shared library
- Link with Samba
- Profit!

Initial Experiment: Building a Shared Library

Explained in the Rust Book

```
[lib]
name = "embed"
crate-type = ["dylib"]
```

Second Experiment: Linking with Samba

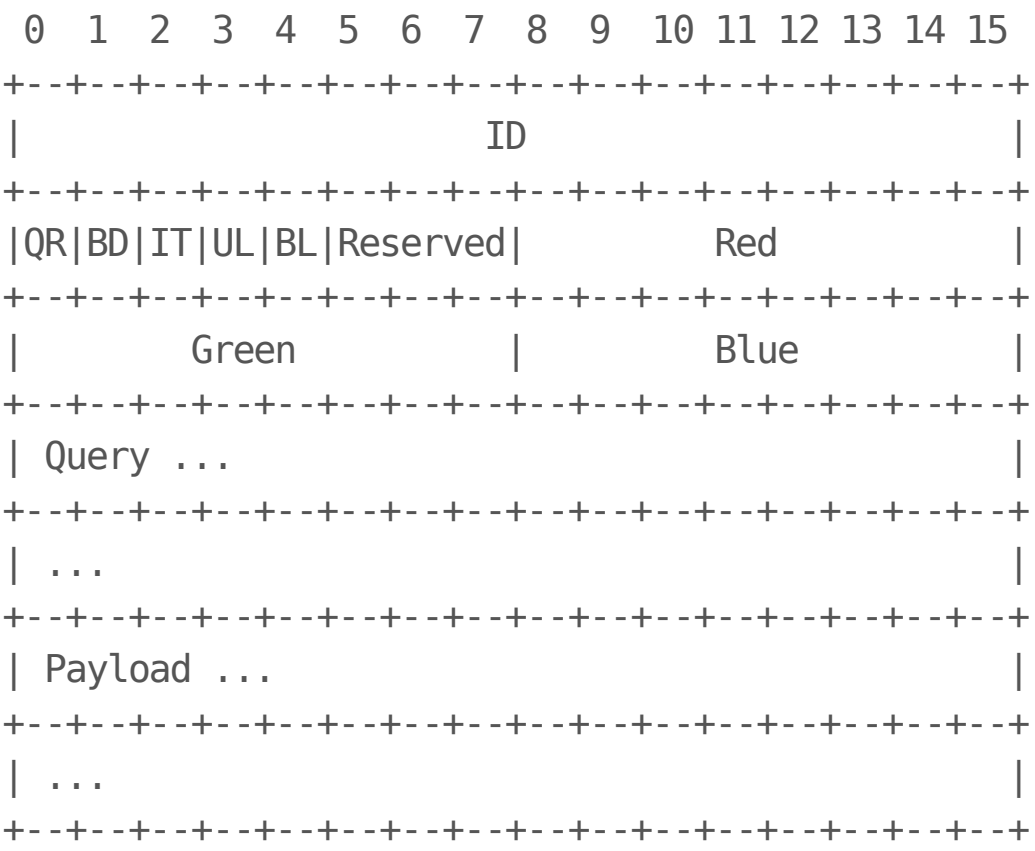
- Preferred build system for Rust is `cargo`
- Preferred build system for Samba is `waf`
- Fiddly and not very exciting

Idea: Build a small demo first, then figure out how to link

New Plan

- Make up a simple DNS-like protocol
- Build a parser for it in Rust
- Build as a shared library
- Load from a C application
- Profit!

The FancyTalk Protocol



The FancyTalk Protocol

- Client sends a query, giving an ID
- Server looks up the query in the database
- Server responds with a payload, using bit flags for formatting and fancy colours

Demo Time!

Server

```
$ cd server
$ cargo run
```

Client

```
$ cd client
$ cargo run 127.0.0.1 65432 greeting
```

In theory, there is no difference between theory and practice. But, in practice, there is.
– Jan L. A. van de Snepscheut

Implementing it

Data structure

```rust
pub struct Package {
    pub id: u16,
    pub message_type: MessageType,
    pub bold: bool,
    pub italic: bool,
    pub underlined: bool,
    pub blink: bool,
    pub red: u8,
    pub green: u8,
    pub blue: u8,
    pub query: Option<String>,
    pub payload: Option<String>,
}

pub enum MessageType {
    Query,
    Response,
}
```

Client

```
let mut query = Package::new();
query.query = Some(config.query);

let mut out_buf: Vec<u8> = Vec::new();
{
    let mut encoder = Encoder::new(&mut out_buf);
    query.write(&mut encoder).expect("Failed to encode query");
}

// send query
// get response

let mut decoder = Decoder::new(&in_buf);
let response = Package::read(&mut decoder).expect("Parsing the response failed");

// Format, colour and print response
```

Rust Server

```
loop {
    // Receive query into inbuf

    let mut decoder = Decoder::new(inbuf);
    let query = Package::read(&mut decoder).expect("Parsing query failed");

    let response = lookup_message(&mut messages, &query);

    let mut outbuf: Vec<u8> = Vec::new();
    {
        let mut encoder = Encoder::new(&mut outbuf);
        response.write(&mut encoder).expect("Encoding response failed");
    }

    // Send response from outbuf
}
```

The C Server Concept

```
while True {
    // Recieve query into in_buffer

    // Call into Rust for parsing in_buf into Package
    query = decode_package(in_buffer, len);

    // "Business logic" in C
    lookup_message(query, response);

    // Call into Rust again to create out_buf for Package
    encode_package(response, &out_buffer, &len);

    // Send response from out_buffer
}
```

The Shared API

```
typedef struct package {
  //...
} Package;

Package *decode_package(const uint8_t* buffer, size_t len);
int encode_package(const Package *package, uint8_t **buffer, size_t *len);
```

Hang on a Moment

Who owns memory for the `Package` struct in `decode_package()`?

- Option 1: Rust
- Option 2: C

Option 1: Rust Owns Memory

- Rust handles memory allocation
- C just uses the structs
- Rust needs to handle deallocation
- C needs to call back into Rust to free memory

Remember the Free Functions

```
typedef struct package {
  //...
} Package;

Package *decode_package(const uint8_t* buffer, size_t len);
int encode_package(const Package *package, uint8_t **buffer, size_t *len);
void free_package(Package *package);
void free_buffer(uint8_t *buffer);
```

- Someone will forget to call the right free soon.

Demo Time!

Server

```
$ cd c-server
$ make run
```

Client

```
$ cd client
$ cargo run 127.0.0.1 6543 greeting
```

Option 2: C Owns Memory

- Memory ownership passed to calling C code
- C takes care of freeing the memory
- Rust needs to allocate memory in a way C can free
- Idea: Port `talloc` to Rust

Detour

Implementing talloc in Rust

Implementing talloc in Rust

- Load `libtalloc` via FFI
- Provide `talloc_named_const()` and `talloc_free()` calls
- Wrap in Rust layer to make this feel like native code.

talloc FFI

```
use libc::{size_t, c_int, c_char, c_void};

#[link(name="talloc")]
extern {
    pub fn talloc_named_const(context: *const c_void, size: size_t,
                              name: *const c_char) -> *mut c_void;
    pub fn _talloc_free(context: *mut c_void,
                        location: *const c_char) -> c_int;
}
```

- Now `talloc` is useable in Rust
- Needs to be wrapped to feel more Rust-like

talloc Rust Datastructures

- Basic datastructure to allocate heap memory is `Box<>`
- Build a `Box<>`-like `TallocContext<>` that uses talloc
- Use that to allocate memory for transferable data structures

talloc Rust Datastructures

```
impl<T> TallocContext<T> {
    pub fn new<U>(parent: Option<TallocContext<U>>) -> TallocContext<T> {
        let size = size_of::<T>();
        let name = CString::new("Memory context allocated from Rust.")
                   .unwrap();
        let parent_ptr = match parent {
            Some(tc) => tc.ptr,
            None => null(),
        };
        unsafe {
            let ptr = ffi::talloc_named_const(parent_ptr as *const c_void,
                                              size as size_t, name.as_ptr());
            TallocContext {
                ptr: ptr as *mut u8,
                phantom: PhantomData
            }
        }
    }
}
```

talloc Rust Datastructures

```
impl<T> Drop for TallocContext<T> {
    fn drop(&mut self) {
        let name = CString::new("Free in the Rust deallocation logic.")
                   .unwrap();
        unsafe {
            let retcode = ffi::_talloc_free(self.ptr as *mut c_void,
                                            name.as_ptr());
            if retcode != 0 {
                panic!("Failed to free memory!");
            }
        };
    }
}
```

There's More

```
pub struct Package {
    pub id: u16,
    pub message_type: MessageType,
    pub bold: bool,
    pub italic: bool,
    pub underlined: bool,
    pub blink: bool,
    pub red: u8,
    pub green: u8,
    pub blue: u8,
    pub query: Option<String>,
    pub payload: Option<String>,
}
```

More Types

```
pub struct TallocString {
    inner: TallocContext<[u8]>,
}

pub struct TallocVec<T> {
    context: TallocContext<T>,
    len: usize,
}
```

- We probably want `talloc` versions of all heap-allocated structures >- Lots of boilerplate

Alternative Idea

- Make `talloc` useable as global allocator in Rust
- Implement `Allocate` trait
- **But**: Not a stable feature yet

End of Detour

Implementing talloc in Rust

Truth is subjectivity.
– Søren Kierkegaard

Conclusions

Conclusions

- Less easy than I had hoped for
- Need to decide which memory handling method to use
- How to integrate build systems?
- How to handle Rust as dependency?
- Rust community is pretty helpful

Future Work

- Better talloc-sys implementation?
- Auto-generate code from IDL
- Build system integration ☹

Thank you