More Fancy Talk about Rust

The allocator strikes back

Kai Blin
Samba Team

SambaXP 2019
2019-06-06

Intro

- M.Sc. in Computational Biology
- Ph.D. in Microbiology
- Samba Team member
- Like to put Samba on small things

Intro

- M.Sc. in Computational Biology
- Ph.D. in Microbiology
- Samba Team member
- Like to put Samba on small things

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

But why again?

- Fell into the memory allocation rabbit hole last year
- Solution I presented wasn't the popular choice afterwards
- More on this in a bit

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!

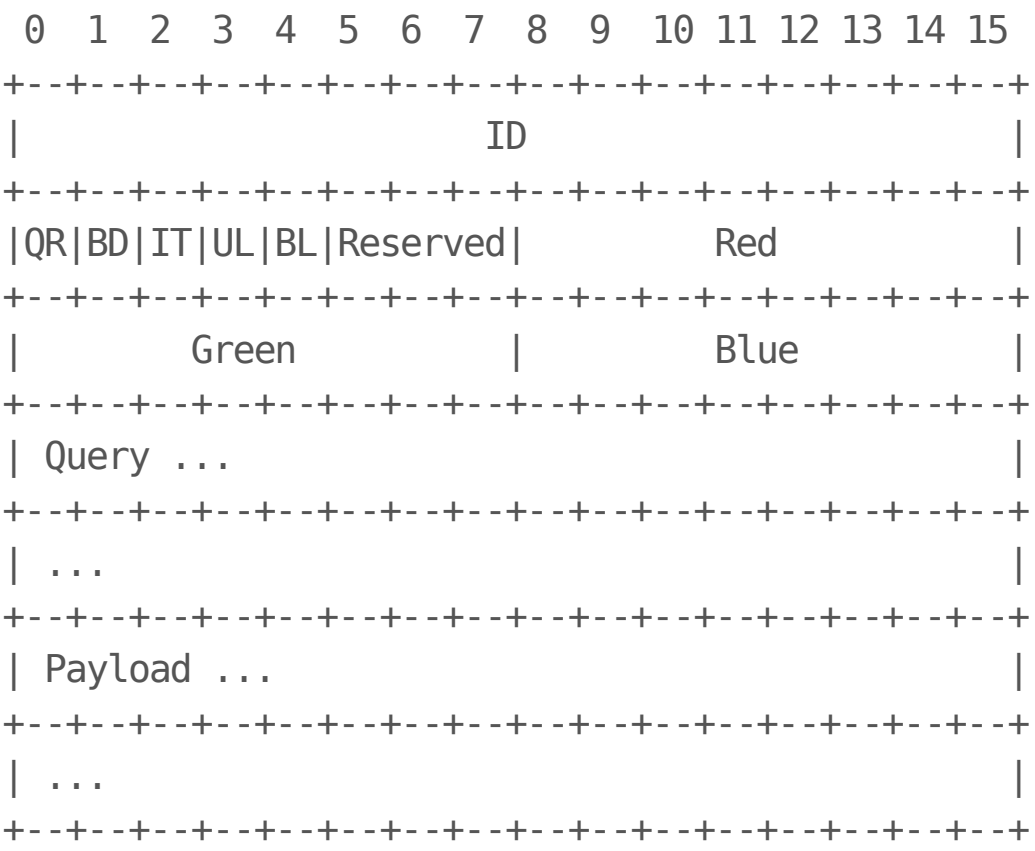
```
fn main() {
    println!("Hello, world!");
}
```

Introducing the example project.

FancyTalk

- A simple DNS-like protocol
- Has a parser built in Rust
- Built as a shared library
- Loaded from a C application

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

```
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.

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

Rabbit Hole

Implementing talloc in Rust

- This is where the project went off the rails
- Maybe let C handle the memory after all

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
- **Idea:** Use talloc destructors

Old version with **malloc**

```
while(1) {
    inbuf = malloc(MAX_UDP_SIZE);
    buflen = recvfrom(...);
    if (buflen == 0) { free(inbuf); continue; }

    query = decode_package((uint8_t *)inbuf, buflen);
    if (query == NULL) { free(inbuf); continue; }

    response = lookup_message(messages, query);

    free_package(query);
    free(inbuf);

    encode_package(response, &outbuf, &buflen);

    buflen = sendto(...);
    free_buffer(outbuf);
}
```

Old version ported to **talloc**

```
while(1) {
    tmp_ctx = talloc_new(mem_ctx);
    inbuf = talloc_size(tmp_ctx, MAX_UDP_SIZE);
    buflen = recvfrom(...);
    if (buflen == 0) { goto done; }

    query = decode_package((uint8_t *)inbuf, buflen);
    if (query == NULL) { goto done; }

    response = lookup_message(messages, query);
    encode_package(response, &outbuf, &buflen);
    sendto(...);
done:
    talloc_free(tmp_ctx);
    free_package(query);
    free_buffer(outbuf, buflen);
}
```

Using `talloc` destructors

Set up

```
struct server_ctx {
    Package *query;
    uint8_t *buffer;
    uintptr_t buflen;
};

int free_server_ctx(struct server_ctx *srv) {
    if (srv->query) {
        free_package(srv->query);
    }
    if (srv->buffer) {
        free_buffer(srv->buffer, srv->buflen);
    }
};
```

Using **talloc** destructors

Main loop

```
while(1) {
    srv_ctx = talloc_zero(mem_ctx, struct server_ctx);
    talloc_set_destructor(srv_ctx, free_server_ctx);
    inbuf = talloc_size(srv_ctx, MAX_UDP_SIZE);
    buflen = recvfrom(...);
    if (buflen == 0) { goto done; }

    srv_ctx->query = decode_package((uint8_t *)inbuf, buflen);
    if (srv_ctx->query == NULL) { goto done; }

    response = lookup_message(messages, srv_ctx->query);
    encode_package(response, &srv_ctx->buffer, &srv_ctx->buflen);
    buflen = sendto(...);
done:
    talloc_free(srv_ctx);
}
```

Demo Time!

Server

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

Client

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

Caveats

- incorrect free functions can still leak memory
- FFI needs lots of `unsafe` blocks
- ideally use opaque pointers for less glue code

Truth is subjectivity.
– Søren Kierkegaard

Conclusions

Conclusions

- How to integrate build systems?
- How to handle Rust as dependency?
- Rust community is pretty helpful, big thanks to mbrubeck, stefaneyfx and matt1992

Future Work

- Auto-generate code from IDL
- Build system integration ☹

Thank you