Asynchronous IO in Rust

Last years of my career I'm building a lot of low-level networking stuff. And most of the time networking stuff should be asynchronous. That's what Rust currently lacks: good asynchronous I/O abstraction. The goal of this article is to bring the larger discussion how it should work.

This article looks a little bit like a brain dump of designed features and motivation behind. The up to date tutorial will be somewhere in the docs. At the time of writing things are moving fast in rust community so many things may become out of date very soon.

Rust has good threading support and since dropping support of libgreen everything in Rust is focused around threads for now. But learning from Java world, we should not rely on threads too much for networking tasks. Also, we don't want to have rust applications having thousands of threads, which is normal in Java. And more importantly it influences the API design. Take a look at hyper, it implements header parsing in a very clean way. But also hyper has complex thread-based machinery that even includes custom buffering, which turns out to be impossible to reuse in an asynchronous fashion.

So what alternatives we have now. There is a very low-level library mio, which does its job well, but the interface is a bit wild. More importantly raw interface it's not composable. I. e. you can't put two different protocol implementations to the same loop and tie them together to build an application. Well, you can put both in different threads but for most use cases it's either too complex, too slow, too inflexible or all at once.

There are also few coroutine libraries and I/O implementation on top of mio. But I don't believe that coroutines might become stable rust soon because of lots of technical issues. Also, while coroutines are great for prototyping things they are not always the best solution for many problems. So I believe we might build something without coroutines on top of mio.

Requirements

First let’s define how we want I/O work in rust:

1. Obviously we want to write only safe code.
2. We don’t want (too many) RefCell’s and Rc’s
3. We don’t want Deferred/Promises because they require too many closures that don’t play nice with rust borrow checker. And they usually turn into “spaghetti code”.
4. We don’t want Agents *at this level of abstraction* because it’s just too expensive to toss bytes through a channel from connection reader to protocol parser to protocol handler, etc.

State Machines Everywhere

So what works well in rust is good old state machines. There are algebraic types in rust that make coding state machine easy and fun.

I.e. here is how state machine for http connection might look like:

```
enum HttpConnection {
    KeepAlive,
    ReadingHeaders(Buffer),
    ReadingBody(Request, Buffer),
    SendingResponse(Buffer),
}
```

Note the following things:

1. Connection buffers are deallocated for keep-alive connection
2. Input and output buffers might occupy same space in connection structure (ReadingHeaders has input buffer -> SendingResponse has output buffer)

The optimization #2 actually isn’t very useful as Request structure might be much bigger than the place buffer occupies in the state machine structure. It can be mitigated too, but I’ll omit this discussion for brevity

The deallocation of buffers is an important optimization, but what made us use this kind of state machine is not actually the optimization. It’s the *intent:* we don't need buffers on a keep-alive connection. We also don't need output buffer when we reading request so we just leave it out of the structure.

The cool property of state machines as algebraic types is that they are usually self-describing compact data structures that not only easy to reason about but also easy to code and very efficient.

So we'll probably build asynchronous code using state machines. Like in good old 1960-x :). But that's not the end of story. First let's look at gory details.

Implementation of State Machine

Let's model some low-level automata:

```
pub trait Protocol: Sized {
  fn data_received(self, data: &str) -> Self;
  fn eof_received(self) -> Self;
}
```

What important here:

- State machine works by consuming "self" by value and returning new "self".
- Input to state machine is provided by calling method

To keep things simpler for executor we may adjust it to something like this:

```
pub trait Protocol: Sized {
  fn data_received(self, data: &str) -> Option<Self>;
  fn eof_received(self);
}
```

In "data_received" the "None" result means the connection should be dropped. It's used so that caller doesn't need to differentiate between final and non-final state of the automata, i.e. final state encoded as "None" and non-final as "Some(_)".

The “eof_received” method returning nothing just communicates to the programmer that there can’t be any state after this signal received.

Okay, That’s Too Simple

Right. So when you know how we will create different kinds of state machines, we are going to discuss more complex things.

First we need a layering. At lowest layer we read and write bytes from/to stream. At a higher level we do framing of the stream. At next level, we have a Request object and Reply object.

The idea to handle it as “hierarchical state machines”. It is hugely inspired by an article of Martin Sustrik, the author of nanomsg and zeromq.

> *A little bit history. The zeromq (if you don’t know what it is you should look at it, it’s very inspiring) project was written in C++ and was based on a state machines and agents. I. e. every TCP connection was an agent. Every “socket” (which is a collection of connections and a little bit of logic in zeromq) is also an agent. And all the objects where somewhat arbitrary distributed between threads in thread pool. It turned out this design is a mess and doesn’t allow complex features to be implemented.*
>
> *The nanomsg that’s basically a rewrite of zeromq, is written in C and is based on some kind of hierarchical state machines. It turned out to be hard to support too for few reasons (solely my own opinion): (a) the type system of C language is not good for state machines [I’ll try to elaborate on the topic below] (b) state machines may be almost freely used between threads (c) socket shutdown code was very racy and complex mostly because of (a) and (b).*

Let’s imagine that HTTP machine looks like the following:

```
enum HttpConnection<H:RequestHandler> {
    KeepAlive,
    ReadingHeaders,
    ReadingBody(H),
    SendingResponse,
}
trait RequestHandler {
    fn headers_received(uri: Uri, method: Method, headers:
Headers)
        -> Option<Self>;
    fn received_body_chunk(self, chunk: &[u8]) ->
Option<Self>;
    fn body_finished(self) -> Response;
}
impl<H> Protocol for HttpConnection<H> { ... }
```

Note the following things:

1. We don't have buffers here because they belong to lower level of hierarchy (it's not the final design, so it might not always be true)
2. When the headers received, we create a RequestHandler. Look at methods in the trait. It's the state machine too.
3. Handler might finish at any moment, presumably resulting in HTTP error 500, but it *must* finish when request is fully received
4. Compare methods of Protocol and RequestHandler. The action of the RequestHandler state machine obviously happens on "data_received". But the owner state machine creates handler only when headers are fully received. That's one of the key points: child state machines receive fewer events, and it implied that the events are at higher level.

> *This is a simplified example; we should have some way to respond asynchronously too, but I skip it for keeping examples shorter*

What rust helps us here:

1. Each state machine owned by the parent. So shutting down any at a lower level (say disconnect happened), will tear down all state machines inside.
2. The Drop trait for state machine obviously work, so if you have any resources open for the state they are cleaned correctly. Ownership also means if you forget to put child state machine in your new state, child machine will shut down and release resources.
3. Rust move semantics allows to work with a state machine as with immutable value (it's hard to break state by modifying only a half of state). Still with the performance of mutable object (i.e. you may move buffers to a new place without reallocating). At this level of abstraction even small performance loss will add up pretty quickly.

Complex Interactions

Well, we discovered three ways of communication between state machines:

1. Sending action to child machine (i.e. invoking a method)
2. Receiving a new state for the child state machine (this is also a way of communication, for example, low-level layers know that connection should not be closed yet)
3. Receiving a value from child machine (i.e. response for the request)

But that communication is inherently local. More things we need:

1. Communication between multiple connections (multiple unrelated state machines) works by sending a `Notify` to the mio channel. Which is basically similar to sending a message in agent-based programming (where each agent is a hierarchical state machine in our case)
2. Creating a state machine. We can't just insert a new connection to the hash map of connections because of borrow checker. So we send the "NewMachine" message to the main loop which creates and registers the machine.
3. Global mutable state. Making every tiny interaction with global state via messaging is very inefficient, so we will discuss global state more in the following sections.
4. Communication between threads. This is out of scope for the article. This is where rust has lots of instruments.

Composition

Now we are ready to the interesting part. The composition.

The composition works by creating a wrapper state machine. Hold on, there is a bit of a boilerplate, but it worth it.

```
struct Left;
struct Right;

enum Wrapper {
    Left(Left),
    Right(Right),
}

impl http::Handler for Left { ... }
impl http::Handler for Right { ... }
impl http::Handler for Wrapper {
    fn request(self, r: Request) -> Option<Self> {
        match self {
            Wrapper::Left(x) =>
x.request(r).map(Wrapper::Left),
            Wrapper::Right(x) =>
x.request(r).map(Wrapper::Right),
        }
    }
}
```

Here we have a “Wrapper” state machine that is just a state machine that delegates all the events to the child state machines, which are declared here as “Left” and “Right”.

At the end of the day it allows:

1. To combine multiple libraries or multiple instances of the the same library
2. Give a chance for compiler to inline methods (no virtual calls)

Frankly, this is not all boilerplate needed. We also need some things for handling global state, more parts of it will be in the following section. And you can look at fully working example. There are also macros that make this simple kind of composition short and trivial.

Global State

Handling global state is a non-trivial problem. Let’s first approach a problem by brute forcing. We could:

1. Pass some context structure to every method. This is basically impossible because we don’t know what structure user will be using when building a library handling a single protocol.
2. Pass an anymap structure as a context. It could work, but we would get runtime errors if something is not in the map. And we want to have better compile-time checking in Rust.
3. Put the global data into static mutable or lazy static variable. But this doesn’t compose well, for example if you want two instances of a protocol handler to coexists in the same process.

Instead we pass a generic context object to every state machine/protocol/library, like this:

```
pub trait Protocol<C>: Sized {
  fn data_recvd(self, data: &str, context: &mut C) ->
Option<Self>;
  fn eof_recvd(self, context: &mut C);
}
```

If any library or programmer want to have a some specific state in a context, it defines a trait and bounds it's implementation to the trait. For example if HTTP library want to count number of simultaneous connections it might do this:

```
trait HTTPContext {
    fn incr_connections();
    fn decr_connections();
    fn get_connections() -> usize;
}
pub trait HTTPHandler<C>: Sized {
    fn handle_req(self, req: Request, ctx: &mut C) ->
Option<Self>;
}
impl<C:HTTPContext> for HTTPHandler<C> {...}
```

This style has the following superior properties:

1. Any number of traits might be implemented on context, so all libraries can be seamlessly combined.
2. It gives the user a freedom to use different HTTP contexts for different bound sockets or use single one (depends on boilerplate when doing composition).
3. You may even share context between threads by doing atomic operations or using mutexes when implementing the trait.
4. And you get compile-time errors if any trait is missing on context.

While trying to get things implemented, I've realized that there is also another kind of global context, a "Scope". It's a thing that allows you to register socket in the event loop or set a timeout. Note that in mio event loop you have only a single "Timeout" type and a single "Message" type. So you can't register "http::Timeout" in event loop and subsequently say "redis::Timeout" that is probably different structure. This is where Scope joins a party. So scope is passed everywhere in addition to context:

```
pub trait Protocol<C>: Sized {
  type Timeout;
  fn data_received<S>(self, data: &str,
    context: &mut C, scope: &mut S)
    -> Option<Self>
    where S: Scope<Self, Self::Timeout>;
}
```

The scope contains "mio::Token" and a reference to "mio::EventLoop" so you can easily add timeouts or change event set in the poll.

For the composition of state machines, you need a little bit boilerplate too. For example to make scope suitable for the "Wrapper" machine above (which consists of "Left" and "Right" options), you do the following:

```
pub struct ScopeLeft<'a, S: 'a>(pub &'a mut S);
```

```
impl <'a, S> Scope<Left, Left::Timeout> for ScopeLeft<'a,
S>
    where S: Scope<Wrapper, TimeoutWrapper> + 'a
{
    fn add_timeout_ms(&mut self, delay: u64, t:
Left::Timeout)
          -> Result<::mio::Timeout, ::mio::TimerError>
    {
        self.0.add_timeout_ms(delay, Wrapper::Left(t))
    }
}
```

Which essentially looks like: wrap the inner type and pass it down the chain. The code for another wrapper type "ScopeRight" and more operations on Scope type are similar, so we omit them for brevity.

> *Note that this kind of structures in rust are actually zero-cost, as they exist only in compile-type. In the run-time, they are equal to the underlying structure.*

Let's summarize difference between context and scope:

1. Context's real type is defined by end application with each library requiring their own traits. Basically, it contains (or references) all data

needed for every library's global bookkeeping.

2. The real struct for Scope is implemented by rotor library itself. When composing applications you only transform types from higher-level ones to lower-level ones and vice versa.

3. The Scope is a lightweight object that is created for each operation and contains a "mio::Token" and pointers to event loop structures.

Unscientific Micro Benchmarks

Just to convince you that state machines and all that complex generics are great, let's run some micro benchmarks. Here is how rust hello world performs on my laptop:

```
> wrk -c 400 -t 2 http://localhost:8888
Running 10s test @ http://localhost:8888
  2 threads and 400 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    10.47ms    3.16ms 241.03ms   97.04%
    Req/Sec    19.25k     2.26k   21.92k    72.50%
  383408 requests in 10.03s, 32.18MB read
Requests/sec:  38232.48
Transfer/sec:      3.21MB
```

The similar hello world server in go performs worse (go 1.5, GOMAXPROCS=1):

```
> wrk -c 400 -t 2 http://localhost:8887
Running 10s test @ http://localhost:8887
  2 threads and 400 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    17.48ms    4.16ms  45.51ms   69.49%
    Req/Sec    11.48k     0.89k   12.82k    78.50%
  228464 requests in 10.03s, 27.89MB read
Requests/sec:  22771.40
Transfer/sec:      2.78MB
```

By the way, no profiling and tuning is done in rust library yet. (However it uses header parsing from hyper that might be well-optimized, I'm not sure).

Conclusion

Designing this stuff was not easy. Especially getting all the generics work (and the job is not fully done yet). But the result looks promising. The code has no Rc's, RefCells, type casts, unsafe code and even no virtual calls, as far as I understand. So it gives the compiler a plenty of room for inlining and other optimizations.

While writing state machines is not always as easy as writing coroutines. It's model is simple enough. It protects from common mistakes like resource leaks and is strongly typed. It also encourages you to cover all possible corner cases. So it's definitely a good trade off for a systems language.

You can try the rotor library and its http (server) implementation now, but be aware that it's interface is in flux now and will break on every occasion until it becomes reasonably good. And if you have any thoughts about the topic, please let me know here, on twitter or the github issues of any projects from above.

Update: hackernews and reddit discussions