Reading From Streams & Writing To Sinks

Ben Sparks

Nix Your Bugs & Rust Your Engines #4

About Me

- Computer Science B.Sc. Magdeburg, M.Sc. Heidelberg
- Junior Software Developer @ Alugha

- First used Rust back in ~2015 (<1.0, think `@T`, `~T`)
- Came back to it in ~2020
- Started using it professionally at Alugha since 12/2023

What this talk is about

0. Refreshers: What are I/O, Streams & Sinks
1. Asynchronous Reading & Writing, Streams & Sinks
2. Applying Codecs to Protocols

Refreshers: I/O, Streams & Sinks

I/O - Blocking vs Asynchronous

- I/O: Input / Output
  - **Reading** from and **writing** to Files, Sockets and more
- Blocking I/O: **Waiting** for I/O operations to finish
  - Further execution of other tasks is **blocked**
- Asynchronous I/O: **Overlap** I/O operations
  - Tasks independent of running I/O operation are **not blocked**

What are Streams & Sinks?

- Streams **emit** potentially infinite data, Sinks **consume** potentially infinite data
- Combined, they form a **data pipeline**
- Supported by many mainstream languages, e.g. C++, Python, Java

	Stream	Sink
Audio	Microphone	Speaker, Headphones
Video	Camera	Display, Storage
Monitoring	Sensor Data	Application Interface
Networking	Incoming TCP Connection	Outgoing TCP Connection

Table 1: Streams and their Sinks

Asynchronous Reading & Writing, Streams & Sinks

Simply Reading & Writing Bytes

- `tokio::io::AsyncRead`: Read bytes asynchronously
- `tokio::io::AsyncWrite`: Write bytes asynchronously
- `tokio::io::split`: Split object into separate handles

	AsyncRead	AsyncWrite
`TcpStream`	✓	✓
`ReadHalf`	✓	X
`WriteHalf`	X	✓

Table 2: Implementors of `Async{Read, Write}`

Reading & Writing Bytes with Freestanding Functions

```
async fn copy<'a, R, W>(r: &'a mut R, w: &'a mut W) -> Result<u64>
```

Listing 1: Unidirectional copying and writing

```
async fn copy_bidirectional<'a, R, W>
(a: &'a mut R + W, b: &'a mut R + W) -> Result<u64>
```

Listing 2: Bidirectional copying and writing

Reading & Writing, Streams & Sinks

- `tokio::io::AsyncRead`: Read bytes asynchronously
- `tokio::io::AsyncWrite`: Write bytes asynchronously

- `futures::Stream<Item = T>`: A stream of values produced asynchronously
- `futures::Sink<Item = T>`: Values can be asynchronously sent into

	Bytes	Item = T
Source	AsyncRead	Stream
Destination	AsyncWrite	Sink

Table 3: Relationship between Asynchronous I/O and Data Pipelines

Transforming between Bytes and Typed Values

```
tokio_util::codec::Decoder: fn(Bytes) -> Result<Option<T>, Error>
tokio_util::codec::Encoder: fn(T) -> Result<Bytes, Error>
```

	Bytes	Codec Half	Item = T
Source	AsyncRead	Decoder	Stream
Destination	AsyncWrite	Encoder	Sink

Table 4: Bridging Bytes and Values

Framed{Read, Write}: Reading & Writing Values with Codecs

- `tokio_util::codec::FramedRead`: **Decode** bytes from `AsyncRead`, emit as values from `Stream`

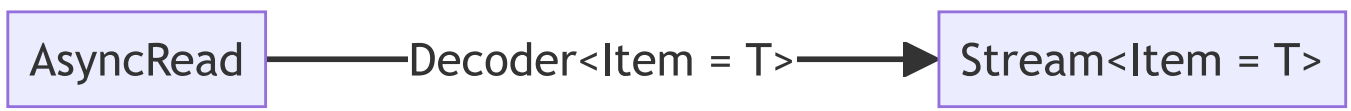

Figure 1: Decoding bytes into values

- `tokio_util::codec::FramedWrite`: **Encode** values, emit as bytes for `AsyncWrite` as a `Sink`

Figure 2: Encoding values as bytes

Decoding and Encoding Protocols with Codecs

Protocols Atop The Transport-Layer: Byte-Based Communication

- Asynchronous I/O: Server can respond independently of the client
- Decoders & Encoders: Interpret commands from their byte representation and vice-versa
- Streams & Sinks: Act upon frames directly to prevent receiving or sending malformed data

- Examples:
  - Pixelflut (Gulaschprogrammiernacht Fun with Colours)
  - NATS (Distributed Messaging)

Pixelflut

Command	Description
HELP	Return a short help text
SIZE	Return size of canvas in pixel as SIZE <w> <h>
PX <x> <y>	Return color of pixel as PX <x> <y> <rrggbb>
PX <x> <y> <rrggbb(aa)>	Blend pixel at position (x, y) with specified colour

- Multiple commands can be sent at once by terminating each one with ‘\n’

- ‘SIZE\n’ → How big is the canvas
- ‘PX 12 34\nPX 56 78\nPX 90 12 AABBCC\n’ → Read twice, write once

Pixelflut ⇄ Bytes

```rust
pub struct PixelflutCodec;

pub enum PixelflutCommand {
  Help, Size, Read(Position), Write(Position, RGBA),
}

impl Decoder for PixelflutCodec {
  type Item = PixelflutCommand;
}

impl Encoder<PixelflutCommand> for PixelflutCodec {
}
```

Listing 3: Decoding & Encoding Pixelflut

Separating Input from Decoding / Output from Encoding

```
// Reading and writing Pixelflut commands to a TcpStream
let (in_, out) = io::split(tcpstream);
let reader = FramedRead::new(in_, PixelflutCodec);
let writer = FramedWrite::new(out, PixelflutCodec);

// Reading and writing Pixelflut commands to a file
let (in_, out) = io::split(fs::File::open(...).await?);
let reader = FramedRead::new(in_, PixelflutCodec);
let writer = FramedWrite::new(out, PixelflutCodec);

// Reading and writing Pixelflut commands to stdin and stdout
let (in_, out) = (io::stdin(), io::stdout());
let reader = FramedRead::new(in_, PixelflutCodec);
let writer = FramedWrite::new(out, PixelflutCodec);
```

NATS

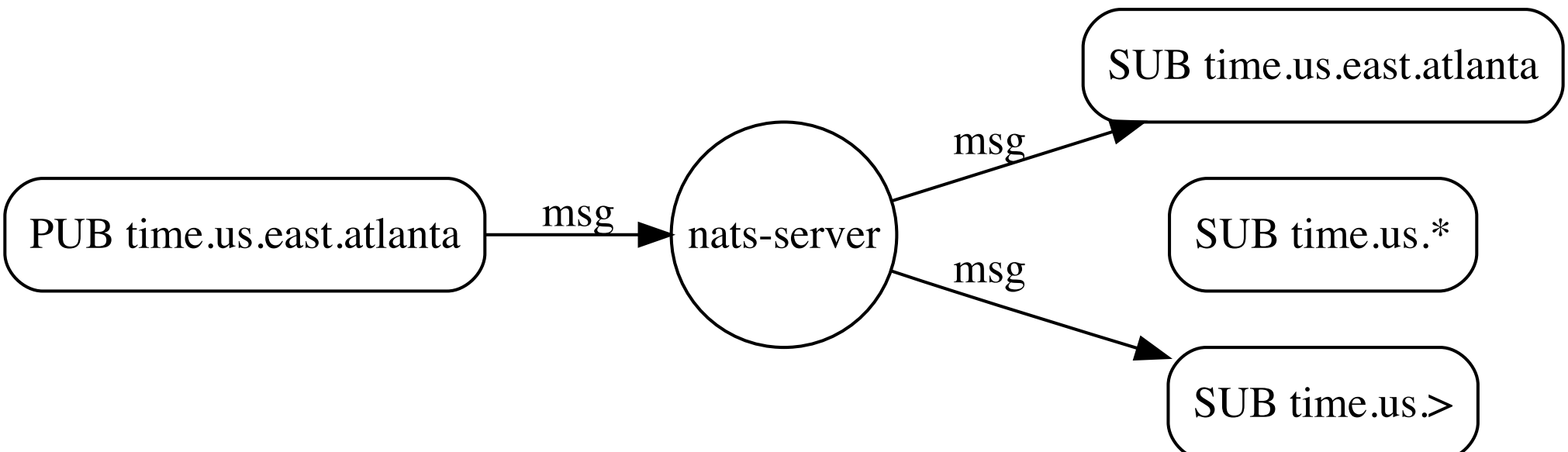

Figure 3: Distributed Messaging with NATS

NATS' Client Protocol

Command	Description
INFO	How to connect to Server?
(H)MSG	Send message to subscriber
+OK/-ERR	Indicate well-formed frame

Table 6: Server ⟶ Client

Command	Description
CONNECT	Provide authentication
(H)PUB	Publish message to subject
(UN)SUB	(Un)Subscribe to / from subject

Table 7: Client ⟶ Server

Command	Description
PING / PONG	Keep-alive message / response

Table 8: Client ⇄ Server

- Client responds to Server's `INFO` with `CONNECT` to establish connection
- Client must implement keep-alive messaging and responding

NATS ⇄ Bytes

```rust
pub struct ClientCodec;

pub enum ClientCommand {
  Connect, Pub, Sub, Unsub,
  Ping, Pong,
}

impl Decoder for ClientCodec {
  type Item = ClientCommand;
}

impl Encoder<ClientCommand> for
ClientCodec { }
```

Listing 6: Client Commands

```rust
pub struct ServerCodec;

pub enum ServerCommand {
  Msg, HMsg, Ok, Err,
  Info, Ping, Pong,
}

impl Decoder for ServerCodec {
  type Item = ServerCommand;
}

impl Encoder<ServerCommand> for
ServerCodec { }
```

Listing 7: Server Commands

Thanks for Listening!